#PostgreSQL#

Section 14. PostgreSQL Data Types in Depth
Boolean – store TRUE and FALSEvalues with the Boolean data type.
CHAR, VARCHAR, and TEXT – learn how to use various character types including CHAR, VARCHAR, and TEXT.
NUMERIC – show you how to use NUMERIC type to store values that precision is required.
DOUBLE PRECISION – learn to store inexact, variable-precision numbers in the database. The DOUBLE PRECISION type is also known as the FLOAT type.
REAL – guide you on how to use single-precision floating-point numbers in the database.
Integer – introduce you to various integer types in PostgreSQL including SMALLINT, INT and BIGINT.
DATE – introduce the DATE data type for storing date values.
Timestamp – understand timestamp data types quickly.
Interval – show you how to use interval data type to handle a period effectively.
TIME – use the TIME datatype to manage the time of day values.
UUID – guide you on how to use UUID datatype and how to generate UUID values using supplied modules.
Array – show you how to work with arrays and introduce you to some handy functions for array manipulation.
hstore – introduce you to the hstore data type, a set of key/value pairs stored in a single value in PostgreSQL.
JSON – illustrate how to work with JSON data type and use some of the most important JSON operators and functions.
User-defined data types – show you how to use the CREATE DOMAIN and CREATE TYPE statements to create user-defined data types.
Enum – learn how to create an enum type that defines a list of fixed values for a column.
XML – show you how to store XML documents in the database using the XML data type.
BYTEA – learn how to store binary strings in the database.
Composite Types – show you how to define a composite type that consists of multiple fields.

Can You Teach Me above all the topics with Realtime example (create a simple table and insert some values) and show output in Table Format and Also Explain Each topic why this needed, what happen, like syntax and so on.

Sure, I can guide you through these PostgreSQL data types one by one. Let's start with **Boolean** data type.

Boolean Data Type

Purpose:

- Stores logical values: `TRUE`, `FALSE`, or `NULL`.
- Useful for flags or conditions (e.g., "Is Active?", "Is Deleted?").

Syntax:

```sql
CREATE TABLE boolean_example (
    id SERIAL PRIMARY KEY,
    is_active BOOLEAN
);

INSERT INTO boolean_example (is_active)
VALUES
    (TRUE),
    (FALSE),
    (NULL);
```

Explanation:

- `BOOLEAN` can only hold `TRUE`, `FALSE`, or `NULL`.
- Use it for columns where binary logic is required.

Output:

After inserting the above values:

```sql
SELECT * FROM boolean_example;
```

id	is_active
1	TRUE
2	FALSE
3	NULL

Would you like me to proceed to the next data type (CHAR, VARCHAR, and TEXT), or do you have questions about this one?

Continue all the topics?

Let's continue with the next topics. I'll provide examples, explanations, syntax, and outputs for each data type.

---

1. CHAR, VARCHAR, and TEXT

Purpose:

- **CHAR(n)**: Fixed-length string.

- **VARCHAR(n)**: Variable-length string with a limit.
- **TEXT**: Variable-length string without a limit.

Syntax:

```sql
CREATE TABLE char_varchar_text_example (
    id SERIAL PRIMARY KEY,
    char_column CHAR(5),
    varchar_column VARCHAR(10),
    text_column TEXT
);

INSERT INTO char_varchar_text_example (char_column, varchar_column, text_column)
VALUES
    ('Hi', 'Hello', 'This is a text column'),
    ('Test', 'Postgres', 'A longer text example'),
    ('Longer', 'Example', 'Learning PostgreSQL');
```

Explanation:

- **CHAR(n)** pads with spaces if the string is shorter than `` `n` ``.
- **VARCHAR(n)** enforces a limit of `` `n` `` characters.
- **TEXT** allows unlimited length.

Output:

```sql
SELECT * FROM char_varchar_text_example;
```

id	char_column	varchar_column	text_column
1	Hi	Hello	This is a text column
2	Test	Postgres	A longer text example
3	Longer	Example	Learning PostgreSQL

2. NUMERIC

Purpose:

- For exact numbers with user-defined precision and scale (e.g., `` `NUMERIC(10, 2)` ``).

Syntax:

```sql
CREATE TABLE numeric_example (
    id SERIAL PRIMARY KEY,
    price NUMERIC(10, 2)
);
```

```
INSERT INTO numeric_example (price)
VALUES
    (123.45),
    (99999.99),
    (0.01);
```

Explanation:

- Suitable for financial and monetary data to avoid rounding errors.

Output:

```sql
SELECT * FROM numeric_example;
```

id	price
1	123.45
2	99999.99
3	0.01

---

3. DOUBLE PRECISION

Purpose:

- Stores approximate, variable-precision floating-point numbers.
- Known as `FLOAT` in other systems.

Syntax:

```sql
CREATE TABLE double_precision_example (
    id SERIAL PRIMARY KEY,
    measurement DOUBLE PRECISION
);

INSERT INTO double_precision_example (measurement)
VALUES
    (123.45678),
    (0.00000001),
    (1.79769e+308);
```

Explanation:

- Use when approximate precision is acceptable (e.g., scientific calculations).

Output:

```sql
sql

SELECT * FROM double_precision_example;
```

id	measurement
1	123.45678
2	0.00000001
3	1.79769e+308

---

4. REAL

Purpose:

- Stores single-precision floating-point numbers.

Syntax:

```sql
sql

CREATE TABLE real_example (
    id SERIAL PRIMARY KEY,
    score REAL
);

INSERT INTO real_example (score)
VALUES
    (98.6),
    (0.123456789),
    (1.5);
```

Explanation:

- Less precise than `DOUBLE PRECISION`.

Output:

```sql
sql

SELECT * FROM real_example;
```

id	score
1	98.6
2	0.123456789
3	1.5

Would you like me to proceed with **Integer types** and others?

Continue?

Let's continue with the next data types.

---

5. Integer (SMALLINT, INT, BIGINT)

Purpose:

- Stores whole numbers of various ranges:
  - **SMALLINT**: -32,768 to 32,767.
  - **INT**: -2,147,483,648 to 2,147,483,647.
  - **BIGINT**: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Syntax:

```sql
CREATE TABLE integer_example (
    id SERIAL PRIMARY KEY,
    small_value SMALLINT,
    int_value INT,
    big_value BIGINT
);

INSERT INTO integer_example (small_value, int_value, big_value)
VALUES
    (32767, 2147483647, 9223372036854775807),
    (-32768, -2147483648, -9223372036854775808),
    (100, 200, 300);
```

Explanation:

- Use appropriate size based on the expected range to save storage.

Output:

```sql
SELECT * FROM integer_example;
```

id	small_value	int_value	big_value
1	32767	2147483647	9223372036854775807
2	-32768	-2147483648	-9223372036854775808
3	100	200	300

---

6. DATE

Purpose:

- Stores calendar dates (year, month, day).

Syntax:

```sql
CREATE TABLE date_example (
    id SERIAL PRIMARY KEY,
    event_date DATE
);

INSERT INTO date_example (event_date)
VALUES
    ('2024-01-01'),
    ('2023-12-31'),
    (CURRENT_DATE);
```

Explanation:

- Use for storing dates without time.
- The `` `CURRENT_DATE` `` function retrieves the current date.

Output:

```sql
SELECT * FROM date_example;
```

id	event_date
1	2024-01-01
2	2023-12-31
3	2024-12-23

---

7. Timestamp

Purpose:

- Stores both date and time, including time zone information if specified.

Syntax:

```sql
CREATE TABLE timestamp_example (
    id SERIAL PRIMARY KEY,
    event_timestamp TIMESTAMP
```

```
);

INSERT INTO timestamp_example (event_timestamp)
VALUES
    ('2024-01-01 12:00:00'),
    ('2023-12-31 23:59:59'),
    (NOW());
```

Explanation:

- Use when both date and time are needed.
- The `` `NOW()` `` function retrieves the current timestamp.

Output:

```sql
SELECT * FROM timestamp_example;
```

id	event_timestamp
1	2024-01-01 12:00:00
2	2023-12-31 23:59:59
3	2024-12-23 10:00:00

---

8. Interval

Purpose:

- Represents a period (e.g., days, months, years).

Syntax:

```sql
CREATE TABLE interval_example (
    id SERIAL PRIMARY KEY,
    time_difference INTERVAL
);

INSERT INTO interval_example (time_difference)
VALUES
    (INTERVAL '1 year 6 months'),
    (INTERVAL '15 days'),
    (INTERVAL '2 hours');
```

Explanation:

- Use for durations or differences in time.

Output:

```sql
SELECT * FROM interval_example;
```

id	time_difference
1	1 year 6 months
2	15 days
3	2 hours

---

9. TIME

Purpose:

- Stores time of day (hours, minutes, seconds).

Syntax:

```sql
CREATE TABLE time_example (
    id SERIAL PRIMARY KEY,
    time_of_day TIME
);

INSERT INTO time_example (time_of_day)
VALUES
    ('08:30:00'),
    ('23:59:59'),
    (CURRENT_TIME);
```

Explanation:

- Use for storing only time values.
- `CURRENT_TIME` retrieves the current system time.

Output:

```sql
SELECT * FROM time_example;
```

id	time_of_day
1	08:30:00
2	23:59:59
3	10:00:00

Would you like me to continue with **UUID**, **Array**, and others?

yes

Let's continue with **UUID**, **Array**, and other data types.

---

10. UUID

Purpose:

- Stores Universal Unique Identifiers (UUIDs).
- Useful for unique identifiers across distributed systems.

Syntax:

```sql
CREATE TABLE uuid_example (
    id SERIAL PRIMARY KEY,
    unique_id UUID
);

-- Insert values
INSERT INTO uuid_example (unique_id)
VALUES
    (gen_random_uuid()), -- Using PostgreSQL's extension
    ('123e4567-e89b-12d3-a456-426614174000'),
    (gen_random_uuid());
```

Explanation:

- `UUID` ensures globally unique values.
- Use `gen_random_uuid()` (requires the `pgcrypto` module).

Output:

```sql
SELECT * FROM uuid_example;
```

id	unique_id
1	e4d909c2-9027-4a93-8d20-6f5f7f321f58
2	123e4567-e89b-12d3-a456-426614174000
3	74b76d88-50b3-4f4e-b098-2c9cf9e6ff44

---

11. Array

Purpose:

- Stores multiple values in a single column.

Syntax:

```sql
sql


CREATE TABLE array_example (
    id SERIAL PRIMARY KEY,
    numbers INTEGER[]
);

-- Insert values
INSERT INTO array_example (numbers)
VALUES
    ('{1,2,3,4,5}'),
    ('{10,20,30}'),
    ('{100}');
```

Explanation:

- Use `**INTEGER[]**` or `**TEXT[]**` for array data.
- Allows array-specific functions like `**array_length**`.

Output:

```sql
sql


SELECT * FROM array_example;
```

id	numbers
1	{1,2,3,4,5}
2	{10,20,30}
3	{100}

---

12. hstore

Purpose:

- Stores key-value pairs in a single column.

Syntax:

```sql
sql


CREATE EXTENSION IF NOT EXISTS hstore;

CREATE TABLE hstore_example (
    id SERIAL PRIMARY KEY,
    data HSTORE
```

```
);

-- Insert values
INSERT INTO hstore_example (data)
VALUES
    ('"key1"=>"value1", "key2"=>"value2"'),
    ('"name"=>"John", "age"=>"30"');
```

Explanation:

- Requires `hstore` extension.
- Useful for semi-structured data.

Output:

```sql
SELECT * FROM hstore_example;
```

id	data
1	"key1"=>"value1", "key2"=>"value2"
2	"name"=>"John", "age"=>"30"

---

13. JSON

Purpose:

- Stores JSON-formatted data.

Syntax:

```sql
CREATE TABLE json_example (
    id SERIAL PRIMARY KEY,
    details JSON
);

-- Insert values
INSERT INTO json_example (details)
VALUES
    ('{"name": "Alice", "age": 25}'),
    ('{"name": "Bob", "age": 30}');
```

Explanation:

- Supports JSON operators and functions for querying and manipulation.

Output:

```sql
SELECT * FROM json_example;
```

id	details
1	{"name": "Alice", "age": 25}
2	{"name": "Bob", "age": 30}

---

14. User-Defined Data Types

Purpose:

- Create custom data types using `CREATE DOMAIN` or `CREATE TYPE`.

Syntax (CREATE DOMAIN):

```sql
CREATE DOMAIN positive_int AS INT
CHECK (VALUE > 0);

CREATE TABLE domain_example (
    id SERIAL PRIMARY KEY,
    positive_number positive_int
);

INSERT INTO domain_example (positive_number)
VALUES (10), (100);
```

Syntax (CREATE TYPE):

```sql
CREATE TYPE mood AS ENUM ('happy', 'sad', 'neutral');

CREATE TABLE type_example (
    id SERIAL PRIMARY KEY,
    mood_column mood
);

INSERT INTO type_example (mood_column)
VALUES ('happy'), ('sad');
```

Explanation:

- **Domains** enforce constraints.
- **Types** allow custom enumerated values.

Output:

```sql
SELECT * FROM domain_example;
```

id	positive_number
1	10
2	100

```sql
SELECT * FROM type_example;
```

id	mood_column
1	happy
2	sad

Let me know if you'd like me to continue with **Enum, XML, BYTEA, and Composite Types**!

yes

Let's continue with **Enum**, **XML**, **BYTEA**, and **Composite Types**.

---

15. Enum

Purpose:

- Stores a list of predefined, fixed values.

Syntax:

```sql
CREATE TYPE color AS ENUM ('red', 'green', 'blue');

CREATE TABLE enum_example (
    id SERIAL PRIMARY KEY,
    favorite_color color
);

-- Insert values
INSERT INTO enum_example (favorite_color)
VALUES ('red'), ('green'), ('blue');
```

Explanation:

- Useful when a column should only have specific values.
- Prevents invalid entries.

Output:

```sql
SELECT * FROM enum_example;
```

id	favorite_color
1	red
2	green
3	blue

---

16. XML

Purpose:

- Stores XML data in a structured format.

Syntax:

```sql
CREATE TABLE xml_example (
    id SERIAL PRIMARY KEY,
    xml_data XML
);

-- Insert values
INSERT INTO xml_example (xml_data)
VALUES
    ('<person><name>John</name><age>30</age></person>'),
    ('<person><name>Jane</name><age>25</age></person>');
```

Explanation:

- Use for applications dealing with XML documents.
- PostgreSQL provides functions to query and manipulate XML.

Output:

```sql
SELECT * FROM xml_example;
```

id	xml_data
1	<person><name>John</name><age>30</age></person>
2	<person><name>Jane</name><age>25</age></person>

17. BYTEA

Purpose:

- Stores binary data, such as files, images, or encrypted data.

Syntax:

```sql
CREATE TABLE bytea_example (
    id SERIAL PRIMARY KEY,
    file_data BYTEA
);

-- Insert binary data (as a string representation for simplicity)
INSERT INTO bytea_example (file_data)
VALUES
    (decode('48656c6c6f20576f726c64', 'hex')), -- "Hello World"
    (decode('4a534f4e206578616d706c65', 'hex')); -- "JSON example"
```

Explanation:

- **`BYTEA`** stores binary strings, often used for files or media.
- The **`decode`** function converts hexadecimal or other encoded data to binary.

Output:

```sql
SELECT id, encode(file_data, 'hex') AS file_data FROM bytea_example;
```

id	file_data
1	48656c6c6f20576f726c64
2	4a534f4e206578616d706c65

18. Composite Types

Purpose:

- Combines multiple fields into a single column.

Syntax:

```sql
CREATE TYPE address AS (
    street VARCHAR,
    city VARCHAR,
    zip_code INT
);

CREATE TABLE composite_example (
    id SERIAL PRIMARY KEY,
    home_address address
);

-- Insert values
INSERT INTO composite_example (home_address)
VALUES
    (ROW('123 Elm St', 'Metropolis', 12345)),
    (ROW('456 Oak St', 'Gotham', 54321));
```

Explanation:

- Allows nesting of related fields in one column.
- Useful for complex data models.

Output:

```sql
SELECT id, (home_address).street AS street, (home_address).city AS city, (home_address).zip_
code AS zip_code
FROM composite_example;
```

id	street	city	zip_code
1	123 Elm St	Metropolis	12345
2	456 Oak St	Gotham	54321

Final Thoughts:

These PostgreSQL data types provide flexibility to handle various data storage needs, from simple numbers and text to complex structures like JSON, XML, and composite types. Each type is designed for specific use cases to ensure data consistency and efficient storage.

Let me know if you'd like detailed explanations, further examples, or help with other PostgreSQL concepts!