**Promt : Give in detailed documentation of create table, different types of data types, various contrraints and syntax. Explain in detail with 2 examples as a beginner in postgreSQL.Also explain about array data type.**

**CREATE TABLE Documentation for PostgreSQL**

The **CREATE TABLE** command is a fundamental part of **Data Definition Language (DDL)** in SQL. DDL commands are used to define and manage the structure of database objects, such as tables, indexes, and views. The **CREATE** statement, in particular, is what you use to create new database objects.

The **CREATE TABLE** command is used to create a new table in a database. A table is a structured collection of data, organized into **rows** and **columns**.

**Syntax**

The basic syntax for creating a table is:

SQL

CREATE TABLE table\_name (

column1\_name data\_type constraint,

column2\_name data\_type constraint,

column3\_name data\_type constraint,

...

columnN\_name data\_type constraint);

* **table\_name:** The name you give to your new table.
* **column\_name:** The name for each column in the table.
* **data\_type:** Specifies the type of data the column can hold (e.g., text, numbers, dates).
* **constraint:** Rules that define how data can be entered into the column (e.g., ensuring a value isn't empty).

**Data Types**

**Data types** are crucial as they determine what kind of values can be stored in a column. Here are some of the most common data types in PostgreSQL:

**Numeric Types**

* **INTEGER:** A whole number with no decimal points. Ideal for IDs or counts.
* **SMALLINT**: A smaller integer type, useful for saving space when the range of numbers is small.
* **BIGINT:** A larger integer type for very large numbers.
* **NUMERIC(p, s):** A number with a user-defined precision. p is the total number of digits, and s is the number of digits after the decimal point. For example, NUMERIC(5, 2) can store a value like 123.45.
* **DOUBLE PRECISION**: A double-precision floating-point number, good for scientific calculations.

**Character Types**

* **VARCHAR(n):** A variable-length string that can store up to n characters. It's the most commonly used string type.
* **TEXT:** A variable-length string with no upper limit. Use this for long-form text like descriptions or blog posts.
* **CHAR(n):** A fixed-length string of n characters. If the input is shorter than n, it's padded with spaces.

**Date/Time Types**

* **DATE:** Stores only the date (e.g., 2025-08-07).
* **TIME:** Stores only the time of day.
* **TIMESTAMP**: Stores both date and time.
* **TIMESTAMPTZ**: Stores date, time, and timezone information.

**Boolean Type**

* **BOOLEAN**: Stores a true or false value.

| **Data Type** | **Description** | **Example Values** |
| --- | --- | --- |
| INT | Integer numbers | 1, 20, -100 |
| BIGINT | Large integers | 99999999999 |
| DECIMAL(p,s) | Fixed-point with precision & scale | 12.34 (DECIMAL(5,2)) |
| FLOAT | Floating-point numbers | 3.14, -10.55 |
| CHAR(n) | Fixed-length string | 'Yes' (CHAR(3)) |
| VARCHAR(n) | Variable-length string | 'John', '[Email@gmail.com](mailto:Email@gmail.com)' |
| TEXT | Long text data | 'This is a long message.' |
| DATE | Date only | '2025-08-07' |
| TIME | Time only | '14:30:00' |
| TIMESTAMP | Date and time | '2025-08-07 14:30:00' |
| BOOLEAN | True or False | TRUE, FALSE |
| SERIAL | Auto-increment integer (PostgreSQL) | 1, 2, 3... |

**Constraints**

**Constraints** are rules that enforce data integrity and prevent invalid data from being inserted into a table.

* **PRIMARY KEY**: A column (or a set of columns) that uniquely identifies each row in the table. A primary key column cannot have a NULL value and must be unique. A table can only have one primary key.
* **FOREIGN KEY**: A column that establishes a link between two tables. It refers to the PRIMARY KEY of another table. This is used to enforce referential integrity.
* **UNIQUE**: Ensures that all values in a column are unique. Unlike PRIMARY KEY, a table can have multiple unique columns, and it can accept one NULL value.
* **NOT NULL:** Ensures that a column cannot have a NULL (empty) value.
* **DEFAULT:** Provides a default value for a column if no value is explicitly provided during an INSERT operation.
* **CHECK:** Enforces a condition on the values of a column. For example, you could use a CHECK constraint to ensure a person's age is greater than 18.

| **Constraint** | **Description** |
| --- | --- |
| PRIMARY KEY | Uniquely identifies each row (NOT NULL + UNIQUE) |
| FOREIGN KEY | Links to primary key of another table |
| UNIQUE | Ensures all values in a column are different |
| NOT NULL | Prevents null (empty) values |
| CHECK | Ensures values meet a condition |
| DEFAULT | Sets a default value |
| AUTO\_INCREMENT | Automatically increases the value (in MySQL) |
| SERIAL | Auto-incrementing (in PostgreSQL) |

**Example 1 – Simple Student Table**

**Create a table to store student information.**

sql

CREATE TABLE students (

student\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

age INT CHECK (age >= 15),

email VARCHAR(100) UNIQUE,

admission\_date DATE DEFAULT CURRENT\_DATE);

**🧾 Explanation:**

* student\_id: Auto-incremented and unique (Primary Key).
* name: Must be provided (NOT NULL).
* age: Must be at least 15.
* email: Must be unique (no duplicates).
* admission\_date: Will use today's date if not provided.

**Example 2 – College Table with Foreign Key**

**Create a college table and link students to it.**

sql

-- **Step 1: Create college table**

CREATE TABLE college (

college\_id SERIAL PRIMARY KEY,

college\_name VARCHAR(100) NOT NULL,

address TEXT);

**-- Step 2: Modify students table to include foreign key**

CREATE TABLE student\_details (

student\_id SERIAL PRIMARY KEY,

name VARCHAR(100) NOT NULL,

age INT,

college\_id INT REFERENCES college(college\_id));

**🧾 Explanation:**

* college\_id: Unique ID for each college.
* college\_name: Required field.
* student\_details.college\_id: Foreign key → references college.college\_id.

**What is ARRAY Data Type in PostgreSQL?**

In PostgreSQL, an **ARRAY** is a **single column** that can hold **multiple values** (of the same data type) inside square brackets [].

Think of it like a **list** inside one cell.  
Example: Instead of having separate rows for a student’s marks, you can store them in a single column as an array:

CopyEdit

{85, 90, 88}

**Why Use Arrays?**

* When one record naturally has **multiple values of the same type**.
* To avoid creating extra tables for small, fixed lists.
* Great for storing things like:
  + Multiple phone numbers
  + Product tags
  + Scores
  + Coordinates

**Syntax for Declaring an Array**

sql

CopyEdit

column\_name data\_type[]

or with size limit:

sql

CopyEdit

column\_name data\_type[array\_size]

**Creating a Table with Array Column**

Example:

sql

CopyEdit

CREATE TABLE students (

id SERIAL PRIMARY KEY,

name VARCHAR(50),

marks INT[]);

Here, marks is an **integer array**.

**Inserting Data into an Array Column**

sql

CopyEdit

INSERT INTO students (name, marks)

VALUES ('Amit', '{85, 90, 88}'),

('Priya', '{78, 82, 89}');

**Note:** Arrays use {} curly braces in PostgreSQL when inserting.

**Retrieving Array Data**

sql

CopyEdit

SELECT name, marks

FROM students;

Output:

pgsql

CopyEdit

name | marks

-------+----------------

Amit | {85,90,88}

Priya | {78,82,89}

**Accessing Specific Elements in an Array**

Arrays in PostgreSQL are **1-indexed** (first element is index 1).

sql

-- Get first mark of each student

SELECT name, marks[1]

FROM students;

Output:

Pgsql

name | marks

-------+-------

Amit | 85

Priya | 78

**Updating an Array Value**

sql

CopyEdit

-- Change the 2nd mark of Amit to 95

UPDATE students

SET marks[2] = 95

WHERE name = 'Amit';

**Useful Array Functions & Operators**

PostgreSQL gives special functions for arrays:

| **Function/Operator** | **Purpose** | **Example** |
| --- | --- | --- |
| ` |  | ` |
| array\_length(arr, 1) | Length of array | array\_length(marks, 1) |
| unnest(arr) | Expand array into rows | SELECT unnest(marks) |
| ANY | Check if a value exists in array | 85 = ANY(marks) |
| ALL | Check if all values match | marks > ALL('{50,60}') |

**Example — Filtering with ANY**

sql

-- Get students who have scored 90 in any subject

SELECT name

FROM students

WHERE 90 = ANY(marks);

**Key Points**

* Arrays should not replace proper **normalization** in big databases — use them for simple, fixed-size multi-value fields.
* You can have **multi-dimensional arrays** (data\_type[][]).
* Arrays work best when combined with **array functions**.