**PostgreSQL - Data Type**

In this chapter, we will discuss the data types used in PostgreSQL. While creating table, for each column, you specify a data type, i.e., what kind of data you want to store in the table fields.

This enables several benefits −

* **Consistency** − Operations against columns of same data type give consistent results and are usually the fastest.
* **Validation** − Proper use of data types implies format validation of data and rejection of data outside the scope of data type.
* **Compactness** − As a column can store a single type of value, it is stored in a compact way.
* **Performance** − Proper use of data types gives the most efficient storage of data. The values stored can be processed quickly, which enhances performance.

PostgreSQL supports a wide set of Data Types. Besides, users can create their own custom data type using *CREATE TYPE* SQL command. There are different categories of data types in PostgreSQL. They are discussed below.

## **Numeric Types**

Numeric types consist of two-byte, four-byte, and eight-byte integers, four-byte and eight-byte floating-point numbers, and selectable-precision decimals. The following table lists the available types.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Storage Size** | **Description** | **Range** |
| smallint | 2 bytes | small-range integer | -32768 to +32767 |
| integer | 4 bytes | typical choice for integer | -2147483648 to +2147483647 |
| bigint | 8 bytes | large-range integer | -9223372036854775808 to 9223372036854775807 |
| decimal | variable | user-specified precision,exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| numeric | variable | user-specified precision,exact | up to 131072 digits before the decimal point; up to 16383 digits after the decimal point |
| real | 4 bytes | variable-precision,inexact | 6 decimal digits precision |
| double precision | 8 bytes | variable-precision,inexact | 15 decimal digits precision |
| smallserial | 2 bytes | small autoincrementing integer | 1 to 32767 |
| serial | 4 bytes | autoincrementing integer | 1 to 2147483647 |
| bigserial | 8 bytes | large autoincrementing integer | 1 to 9223372036854775807 |

## **Monetary Types**

The *money* type stores a currency amount with a fixed fractional precision. Values of the *numeric, int, and bigint* data types can be cast to *money*. Using Floating point numbers is not recommended to handle money due to the potential for rounding errors.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Storage Size** | **Description** | **Range** |
| money | 8 bytes | currency amount | -92233720368547758.08 to +92233720368547758.07 |

## **Character Types**

The table given below lists the general-purpose character types available in PostgreSQL.

|  |  |
| --- | --- |
| **S. No.** | **Name & Description** |
| 1 | **character varying(n), varchar(n)**  variable-length with limit |
| 2 | **character(n), char(n)**  fixed-length, blank padded |
| 3 | **text**  variable unlimited length |

## **Binary Data Types**

The *bytea* data type allows storage of binary strings as in the table given below.

|  |  |  |
| --- | --- | --- |
| **Name** | **Storage Size** | **Description** |
| bytea | 1 or 4 bytes plus the actual binary string | variable-length binary string |

## **Date/Time Types**

PostgreSQL supports a full set of SQL date and time types, as shown in table below. Dates are counted according to the Gregorian calendar. Here, all the types have resolution of **1 microsecond / 14 digits** except **date** type, whose resolution is **day**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Storage Size** | **Description** | **Low Value** | **High Value** |
| timestamp [(p)] [without time zone ] | 8 bytes | both date and time (no time zone) | 4713 BC | 294276 AD |
| TIMESTAMPTZ | 8 bytes | both date and time, with time zone | 4713 BC | 294276 AD |
| date | 4 bytes | date (no time of day) | 4713 BC | 5874897 AD |
| time [ (p)] [ without time zone ] | 8 bytes | time of day (no date) | 00:00:00 | 24:00:00 |
| time [ (p)] with time zone | 12 bytes | times of day only, with time zone | 00:00:00+1459 | 24:00:00-1459 |
| interval [fields ] [(p) ] | 12 bytes | time interval | -178000000 years | 178000000 years |

## **Boolean Type**

PostgreSQL provides the standard SQL type Boolean. The Boolean data type can have the states *true*, *false*, and a third state, *unknown*, which is represented by the SQL null value.

|  |  |  |
| --- | --- | --- |
| **Name** | **Storage Size** | **Description** |
| boolean | 1 byte | state of true or false |

## **Enumerated Type**

Enumerated (enum) types are data types that comprise a static, ordered set of values. They are equivalent to the enum types supported in a number of programming languages.

Unlike other types, Enumerated Types need to be created using CREATE TYPE command. This type is used to store a static, ordered set of values. For example compass directions, i.e., NORTH, SOUTH, EAST, and WEST or days of the week as shown below −

CREATE TYPE week AS ENUM ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun');

Enumerated, once created, can be used like any other types.

## **Geometric Type**

Geometric data types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Storage Size** | **Representation** | **Description** |
| point | 16 bytes | Point on a plane | (x,y) |
| line | 32 bytes | Infinite line (not fully implemented) | ((x1,y1),(x2,y2)) |
| lseg | 32 bytes | Finite line segment | ((x1,y1),(x2,y2)) |
| box | 32 bytes | Rectangular box | ((x1,y1),(x2,y2)) |
| path | 16+16n bytes | Closed path (similar to polygon) | ((x1,y1),...) |
| path | 16+16n bytes | Open path | [(x1,y1),...] |
| polygon | 40+16n | Polygon (similar to closed path) | ((x1,y1),...) |
| circle | 24 bytes | Circle | <(x,y),r> (center point and radius) |

## **Network Address Type**

PostgreSQL offers data types to store IPv4, IPv6, and MAC addresses. It is better to use these types instead of plain text types to store network addresses, because these types offer input error checking and specialized operators and functions.

|  |  |  |
| --- | --- | --- |
| **Name** | **Storage Size** | **Description** |
| cidr | 7 or 19 bytes | IPv4 and IPv6 networks |
| inet | 7 or 19 bytes | IPv4 and IPv6 hosts and networks |
| macaddr | 6 bytes | MAC addresses |

## **Bit String Type**

Bit String Types are used to store bit masks. They are either 0 or 1. There are two SQL bit types: **bit(n)** and **bit varying(n)**, where n is a positive integer.

## **Text Search Type**

This type supports full text search, which is the activity of searching through a collection of natural-language documents to locate those that best match a query. There are two Data Types for this −

|  |  |
| --- | --- |
| **S. No.** | **Name & Description** |
| 1 | **tsvector**  This is a sorted list of distinct words that have been normalized to merge different variants of the same word, called as "lexemes". |
| 2 | **tsquery**  This stores lexemes that are to be searched for, and combines them honoring the Boolean operators & (AND), | (OR), and ! (NOT). Parentheses can be used to enforce grouping of the operators. |

## **UUID Type**

A UUID (Universally Unique Identifiers) is written as a sequence of lower-case hexadecimal digits, in several groups separated by hyphens, specifically a group of eight digits, followed by three groups of four digits, followed by a group of 12 digits, for a total of 32 digits representing the 128 bits.

An example of a UUID is − 550e8400-e29b-41d4-a716-446655440000

## **XML Type**

The XML data type can be used to store XML data. For storing XML data, first you have to create XML values using the function xmlparse as follows −

XMLPARSE (DOCUMENT '<?xml version="1.0"?>

<tutorial>

<title>PostgreSQL Tutorial </title>

<topics>...</topics>

</tutorial>')

XMLPARSE (CONTENT 'xyz<foo>bar</foo><bar>foo</bar>')

## **JSON Type**

The *json* data type can be used to store JSON (JavaScript Object Notation) data. Such data can also be stored as *text*, but the *json* data type has the advantage of checking that each stored value is a valid JSON value. There are also related support functions available, which can be used directly to handle JSON data type as follows.

|  |  |
| --- | --- |
| **Example** | **Example Result** |
| array\_to\_json('{{1,5},{99,100}}'::int[]) | [[1,5],[99,100]] |
| row\_to\_json(row(1,'foo')) | {"f1":1,"f2":"foo"} |

## **Array Type**

PostgreSQL gives the opportunity to define a column of a table as a variable length multidimensional array. Arrays of any built-in or user-defined base type, enum type, or composite type can be created.

### **Declaration of Arrays**

Array type can be declared as

CREATE TABLE monthly\_savings (

name text,

saving\_per\_quarter integer[],

scheme text[][]

);

or by using the keyword "ARRAY" as

CREATE TABLE monthly\_savings (

name text,

saving\_per\_quarter integer ARRAY[4],

scheme text[][]

);

### **Inserting values**

Array values can be inserted as a literal constant, enclosing the element values within curly braces and separating them by commas. An example is shown below −

INSERT INTO monthly\_savings

VALUES (‘Manisha’,

‘{20000, 14600, 23500, 13250}’,

‘{{“FD”, “MF”}, {“FD”, “Property”}}’);

### **Accessing Arrays**

An example for accessing Arrays is shown below. The command given below will select the persons whose savings are more in second quarter than fourth quarter.

SELECT name FROM monhly\_savings WHERE saving\_per\_quarter[2] > saving\_per\_quarter[4];

### **Modifying Arrays**

An example of modifying arrays is as shown below.

UPDATE monthly\_savings SET saving\_per\_quarter = '{25000,25000,27000,27000}'

WHERE name = 'Manisha';

or using the ARRAY expression syntax −

UPDATE monthly\_savings SET saving\_per\_quarter = ARRAY[25000,25000,27000,27000]

WHERE name = 'Manisha';

### **Searching Arrays**

An example of searching arrays is as shown below.

SELECT \* FROM monthly\_savings WHERE saving\_per\_quarter[1] = 10000 OR

saving\_per\_quarter[2] = 10000 OR

saving\_per\_quarter[3] = 10000 OR

saving\_per\_quarter[4] = 10000;

If the size of array is known, the search method given above can be used. Else, the following example shows how to search when the size is not known.

SELECT \* FROM monthly\_savings WHERE 10000 = ANY (saving\_per\_quarter);

## **Composite Types**

This type represents a list of field names and their data types, i.e., structure of a row or record of a table.

### **Declaration of Composite Types**

The following example shows how to declare a composite type

CREATE TYPE inventory\_item AS (

name text,

supplier\_id integer,

price numeric

);

This data type can be used in the create tables as below −

CREATE TABLE on\_hand (

item inventory\_item,

count integer

);

### **Composite Value Input**

Composite values can be inserted as a literal constant, enclosing the field values within parentheses and separating them by commas. An example is shown below −

INSERT INTO on\_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);

This is valid for the *inventory\_item* defined above. The ROW keyword is actually optional as long as you have more than one field in the expression.

### **Accessing Composite Types**

To access a field of a composite column, use a dot followed by the field name, much like selecting a field from a table name. For example, to select some subfields from our on\_hand example table, the query would be as shown below −

SELECT (item).name FROM on\_hand WHERE (item).price > 9.99;

You can even use the table name as well (for instance in a multitable query), like this −

SELECT (on\_hand.item).name FROM on\_hand WHERE (on\_hand.item).price > 9.99;

## **Range Types**

Range types represent data types that uses a range of data. Range type can be discrete ranges (e.g., all integer values 1 to 10) or continuous ranges (e.g., any point in time between 10:00am and 11:00am).

The built-in range types available include the following ranges −

* **int4range** − Range of integer
* **int8range** − Range of bigint
* **numrange** − Range of numeric
* **tsrange** − Range of timestamp without time zone
* **tstzrange** − Range of timestamp with time zone
* **daterange** − Range of date

Custom range types can be created to make new types of ranges available, such as IP address ranges using the inet type as a base, or float ranges using the float data type as a base.

Range types support inclusive and exclusive range boundaries using the [ ] and ( ) characters, respectively. For example '[4,9)' represents all the integers starting from and including 4 up to but not including 9.

## **Object Identifier Types**

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. If *WITH OIDS* is specified or *default\_with\_oids* configuration variable is enabled, only then, in such cases OIDs are added to user-created tables. The following table lists several alias types. The OID alias types have no operations of their own except for specialized input and output routines.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **References** | **Description** | **Value Example** |
| oid | any | numeric object identifier | 564182 |
| regproc | pg\_proc | function name | sum |
| regprocedure | pg\_proc | function with argument types | sum(int4) |
| regoper | pg\_operator | operator name | + |
| regoperator | pg\_operator | operator with argument types | \*(integer,integer) or -(NONE,integer) |
| regclass | pg\_class | relation name | pg\_type |
| regtype | pg\_type | data type name | integer |
| regconfig | pg\_ts\_config | text search configuration | English |
| regdictionary | pg\_ts\_dict | text search dictionary | simple |

## **Pseudo Types**

The PostgreSQL type system contains a number of special-purpose entries that are collectively called pseudo-types. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type.

The table given below lists the existing pseudo-types.

|  |  |
| --- | --- |
| **S. No.** | **Name & Description** |
| 1 | **any**  Indicates that a function accepts any input data type. |
| 2 | **anyelement**  Indicates that a function accepts any data type. |
| 3 | **anyarray**  Indicates that a function accepts any array data type. |
| 4 | **anynonarray**  Indicates that a function accepts any non-array data type. |
| 5 | **anyenum**  Indicates that a function accepts any enum data type. |
| 6 | **anyrange**  Indicates that a function accepts any range data type. |
| 7 | **cstring**  Indicates that a function accepts or returns a null-terminated C string. |
| 8 | **internal**  Indicates that a function accepts or returns a server-internal data type. |
| 9 | **language\_handler**  A procedural language call handler is declared to return language\_handler. |
| 10 | **fdw\_handler**  A foreign-data wrapper handler is declared to return fdw\_handler. |
| 11 | **record**  Identifies a function returning an unspecified row type. |
| 12 | **trigger**  A trigger function is declared to return trigger. |
| 13 | **void**  Indicates that a function returns no value. |