POSTGRESQL - DATA TYPE

http://www.tutorialspoint.com/postgresql/postgresql_data_types.htm Copyright © tutorialspoint.com

This chapter discusses PostgreSQL Data Types. 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 the 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 as 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. Table below 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 below lists general-purpose character types available in PostgreSQL.

Name	Description
character varying*n*, varchar*n*	variable-length with limit
character*n*, char*n*	fixed-length, blank padded
text	variable unlimited length

Binary Data Types

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

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

Date/Time Types

PostgreSQL supports the 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 *notimezone*	4713 BC	294276 AD
timestamp [*p*] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD
date	4 bytes	date *notimeofday*	4713 BC	5874897 AD
time [*p*] [without time zone]	8 bytes	time of day *nodate*	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 type can have several 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 below:

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

Enumerated once created, they 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 *notfullyimplemented*	(*x*1, *y*1,*x*2, *y*2)
lseg	32 bytes	Finite line segment	(*x*1, *y*1,*x*2, *y*2)
box	32 bytes	Rectangular box	(*x*1, *y*1,*x*2, *y*2)
path	16+16n bytes	Closed path *similartopolygon*	(*x*1, *y*1,...)
path	16+16n bytes	Open path	[*x*1, *y*1,...]
polygon	40+16n	Polygon *similartoclosedpath*	(*x*1, *y*1,...)
circle	24 bytes	Circle	<*x*, *y*,r> *centerpointandradius*

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 :

Name	Description
tsvector	This is a sorted list of distinct words that have been normalized to merge different variants of the same word, called as "lexemes".
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 $UniversallyUniqueIdentifiers$ is written as a sequence of lower-case hexadecimal digits, in several groups separated by hyphens, specifically a group of 8 digits followed by three groups of 4 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 create XML values using 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 $JavaScriptObjectNotation$ 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 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 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 as 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 below will select 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 above search method can be used. Else, the following example shows how to search when 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 as 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 $forinstanceinamultitablequery$, like this:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

Range Types

Range types represent data type that uses a range of data. Range type can be discrete ranges $e.g., allintegervalues1to10$ or continuous ranges $e.g., anypointintimebetween10:00amand11:00am$.

The built-in range types available include 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, e.g., '[4,9]' represents all 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 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 below lists the existing pseudo-types.

Name	Description
any	Indicates that a function accepts any input data type.
anyelement	Indicates that a function accepts any data type.
anyarray	Indicates that a function accepts any array data type.
anynonarray	Indicates that a function accepts any non-array data type.
anyenum	Indicates that a function accepts any enum data type.
anyrange	Indicates that a function accepts any range data type.
cstring	Indicates that a function accepts or returns a null-terminated C string.

internal	Indicates that a function accepts or returns a server-internal data type.
language_handler	A procedural language call handler is declared to return language_handler.
fdw_handler	A foreign-data wrapper handler is declared to return fdw_handler.
record	Identifies a function returning an unspecified row type.
trigger	A trigger function is declared to return trigger.
void	Indicates that a function returns no value.

Processing math: 100%