2. Data Types in PostgreSQL

Contents

[1. Course Overview 1](#_Toc25063)

# 1. Introduction

=>slides: Pg. 1

PostgreSQL is one of the world's most advanced and most widely used open-source databases. Over the years, numerous data types and functions, developed by a worldwide team of volunteers, have been added to PostgreSQL. In this module, you will learn about essential data types and functions available in PostgreSQL and see a demonstration of their use cases, which will help you get better at database development and data analysis tasks.

# Data Types in PostgreSQL

=>slides: Pg. 2

In PostgreSQL, there are multiple aliases for most data types in the form of short or long keywords, and optional attributes like length, precision, and scale can be mentioned.

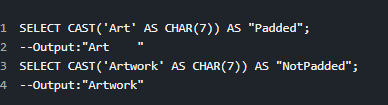
### Character

The most elementary data type used to store character values is known as CHAR or CHARACTER. A fixed length for the character value has to be defined along with the keyword as CHAR(n) or CHARACTER(n). In case a character value of smaller length is added, it is padded with white space for the remaining characters.

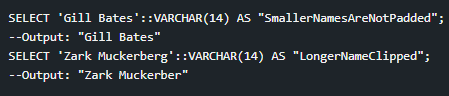
SELECT 'Art'::CHAR(7) AS "Padded";2--Output:"Art

"3SELECT 'Artwork'::CHAR(7) AS "NotPadded";4--Output:"Artwork"

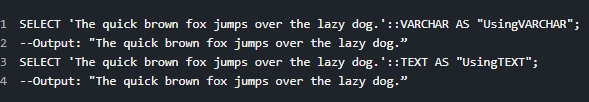
The :: operator is used for explicit type casting as per the syntax in PostgreSQL. It can also be achieved by using the Cast() function.



A good use case for fixed character data types like CHAR is columns where certain values or codes of fixed length are stored, such as Postal Codes. For character values of a varying length, the datatype VARCHAR(n) or CHARACTER VARYING(n) or CHAR VARYING(n) is used. The length can vary up to the maximum length defined by n. The character values smaller than the maximum length are not padded in this datatype.



In case of casting, the longer values get clipped, but inserting longer data values than defined in the column definition will result in an error. For columns where the maximum length cannot be defined due to the extremely variable nature of character values, the maximum length can be omitted in the syntax along with the keyword. There is also a TEXT datatype that is specifically used in such cases.



### Numeric

For storing numbers without fractional components, there are three data types available in PostgreSQL:

* SMALLINT or INT2

This is used if only small range integer values are to be stored and must be preferred only if there are constraints on disk space. It takes up two bytes of storage.

* INTEGER or INT or INT4

This is the most commonly used and preferred datatype since it provides the best balance between range, storage size, and performance. It takes up 4 bytes of storage.

* BIGINT or INT8

In cases where the range of the INTEGER datatype is not sufficient and larger integer values are to be stored, BIGINT is the preferred datatype. It takes up eight bytes of storage.

For numeric values with fixed precision, the NUMERIC or DECIMAL datatype can be used. The length and precision attributes are mentioned along with the NUMERIC or DECIMAL keyword.

1SELECT 9.99::NUMERIC(8,3) AS "NumericValue";2--Output: 9.990

In case they are omitted, numeric values of any precision and scale can be stored in the column up to the implementation limit.

1SELECT 999989999999.889999::NUMERIC AS "NumericValue";2--Output: 999989999999.889999

The value to be stored is rounded to the specified number of fractional digits in case the value is greater than the declared scale.

1SELECT 99998.889999::NUMERIC(8,3) AS "NumericValue";2--Output: 99998.890

Apart from the ordinary numeric values, the numeric type allows the special value NaN (i.e. Not-a-Number).

1SELECT 'NaN'::NUMERIC AS "Not-a-Number";2--Output: NaN

If the values to be stored may have variable precision, they are stored as approximations through the use of following floating-point data types:

* REAL (or FLOAT4)

1SELECT 99998999.889999::REAL AS "RealValue";2--Output: 9.9999e+007

* DOUBLE PRECISION (or FLOAT8)

1SELECT 99998999.889999::DOUBLE PRECISION AS "DoublePrecisionValue";2--Output: 99998999.889999

Note: Storing and retrieving such values might result in slight discrepancies depending on the underlying processor, operating system, and compiler.

For storing monetary values, the MONEY datatype can be used. It allows storage of currency symbols and commas.

1SELECT '$100,000.00'::MONEY AS "CTC";2--Output: "$100,000.00"

### Boolean

At times there could be a requirement to store TRUE, FALSE, or UNKOWN flags in a particular column. For such a case, the BOOLEAN datatype can be used. TRUE/FALSE values can be represented as Y/N, T/F, 1/0 and ON/OFF as well. UNKOWN values are represented by NULL.

SELECT 1::BOOLEAN AS "TrueValue", 0::BOOLEAN AS "FalseValue",NULL::BOOL AS "Unknown";

2--Output: true false [null]

=>slides: Pg. 3

### Date/Time

* DATE This is used to store date values without a time of day.

1SELECT '2020, 4 July'::DATE AS "US Independence Day";2--Output:"2020-04-07"3SELECT '20200407'::DATE AS " US Independence Day";4--Output:"2020-04-07"

* TIME This is used to store only time of day values, which can be with or without the time zone.

1SELECT '12:00:00.12345678'::TIME(2) AS "Noon with 2 micro-second precision";2--Output: "12:00:00.12"34SELECT '00:00:00 IST'::TIME WITH TIME ZONE AS "Midnight as per Indian Standard Time";5--Output: "00:00:00+02:00"

* TIMESTAMP This is used to store the date along with the time of day.

1SELECT '2020, 4 July 12:00:00'::TIMESTAMP AS "US Independence Day NOON";2--Output: "2020-07-04 12:00:00"3SELECT '2020, 4 July 12:00:00 EST'::TIMESTAMP WITH TIME ZONE AS "US Independence Day NOON with Time zone";4--Output: "2020-07-04 22:30:00+05:30"

* INTERVAL This is a quite versatile datatype that can accept date/time-based intervals in various formats.

1SELECT '1 Y 5 Mon 2 H 5 M'::INTERVAL AS "1Year2Months2Hours5Minutes";2--Output:"1 year 5 mons 02:05:00"3select '11 hours, 30 minutes, 40 seconds ago'::INTERVAL AS "11 and a half hours, 40 seconds ago";4--Output: "-11:30:40"

### Array

If there is a requirement to store a collection of values belonging to an in-built or user-defined datatype, an ARRAY can be defined by simply adding square brackets along with the keyword for the datatype.

1select '{ "Certification practice exams", "Interactive courses", "Technical Projects" }'::VARCHAR[] as "Pluralsight Premium"2--Output: "{"Certification practice exams","Interactive courses","Technical Projects"}"3SELECT ARRAY[55,66,77]::INTEGER[] AS "Integer Array"4--Output: {55,66,77}

Unlike programming languages, the array index in PostgreSQL begins with 1 rather than 0.

1SELECT (ARRAY[55,66,77]::INTEGER[])[1] AS "First Array Element"2--Output: 55

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*