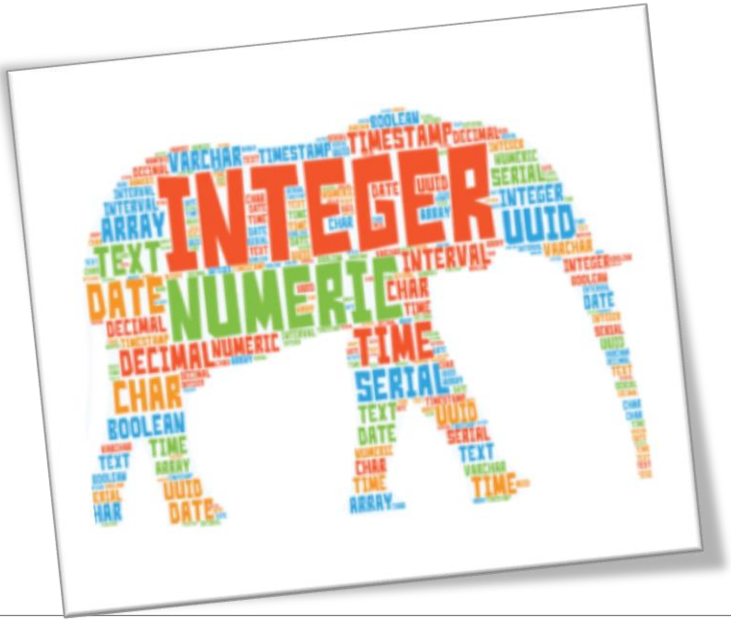
Advanced Database



COMP412

CHAPTER 05: DATA TYPES

Introduction

- PostgreSQL supports the workhorse data types of any database: numerics, strings, dates, times, and booleans.

- PostgreSQL adds support also for arrays, time zone, datetimes, time intervals, ranges, JSON, XML, and many more.

- If that's not enough, you can invent custom types.

Numerics

Numeric

- You will find your everyday integers, decimals, and floating-point numbers in PostgreSQL.
  - Ex: NUMERIC (precision, scale)

Parameter	Description
Numeric	It is a keyword, which is used to store the numeric numbers.
Precision	It is the total number of digits
Scale	It is several digits in terms of the fraction part.

  - Suppose we have the number 2356.78. In this number, the precision is 6, and the scale is 2.
  - Of the numeric types, we want to discuss serial data types.

Serials

- Serial and bigserial are auto-incrementing integers often used as primary keys of tables in which a natural key is not apparent.

- When you create a table and specify a column as serial, PostgreSQL first creates an integer column and then creates a sequence object named table_name_column_name_seq located in the same schema as the table.
  - It then sets the default of the new integer column to read its value from the sequence.

- If you drop the column, PostgreSQL also drops the companion sequence object.

Sequence type

- You can inspect and edit the sequences using SQL with the ALTER SEQUENCE command or using PGAdmin.
- You can set the current value, boundary values (both the upper and lower bounds), and even how many numbers to increment each time.
- Though decrementing is rare, you can do it by setting the increment value to a negative number.

Sequence type

- Because sequences are independent database assets, you can create them separately from a table using the CREATE SEQUENCE command, and you can use the same sequence across multiple tables.
- The cross-table sharing of the same sequence comes in handy when you're assigning a universal key in your database.

```
CREATE SEQUENCE s START 1;
CREATE TABLE stuff(id bigint DEFAULT nextval('s') PRIMARY KEY, name text);
```

Textuals

Textuals

- There are three primitive textual types in PostgreSQL:
  - character (abbreviable as char),
  - character varying (abbreviable as varchar),
  - text.

Textuals

- Use char only when the values stored are fixed length, such as
  - postal codes, phone numbers, and Social Security numbers in the US.
- If your value is under the length specified, PostgreSQL automatically adds spaces to the end.
- When compared with varchar or text, the right-padding takes up more superfluous storage, but you get the assurance of an invariable length.
- There is absolutely no speed performance benefit of using char over varchar or text and char will always take up more disk space.

Textuals

- Use character varying to store strings with varying length.
- When defining varchar columns, you should specify the maximum length of a varchar.
- Text is the most generic of the textual data types. With text, you cannot specify a maximum length.
- The max length modifier for varchar is optional. Without it, varchar behaves almost identically to text.
- Both varchar and text have a maximum storage of 1G for each value

String Functions

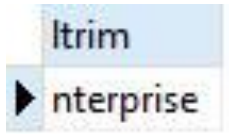
- Common string manipulations are
  - padding (lpad, rpad),
    - The PostgreSQL LPAD() function pads a string on the left to a specified length with a sequence of characters.

```
SELECT LPAD('PostgreSQL',15,'*');
*****PostgreSQL
```

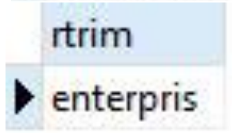
String Functions

- Common string manipulations are
  - trimming whitespace (rtrim, ltrim, trim, btrim)
    - The PostgreSQL rtrim function is used to remove spaces(if no character(s) is provided as trimming_text)

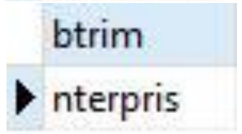
```
SELECT LTRIM('enterprise', 'e');

SELECT RTRIM('enterprise', 'e');

SELECT BTRIM('enterprise', 'e');
```



```
ltrim
nterprise
```



```
rtrim
enterpris
```



```
btrim
nterpris
```

Temporals

Temporals

- In addition to the usual dates and times types, PostgreSQL supports time zones, enabling the automatic handling of daylight saving time (DST) conversions by region.
- At last count, PostgreSQL has nine temporal data types. If
- a type is time zone–aware, the time changes if you change your server’s time zone.

Temporals

- The main types are:
  - Date: Stores the month, day, and year, with no time zone
  - Time: Stores hours, minutes, and seconds with no awareness of time zone
  - Timestamp: Stores both calendar dates and time (hours, minutes, seconds) but does not care about the time zone.
  - Timestamptz: A time zone–aware date and time data type. Internally, timestamptz is stored in Coordinated Universal Time (UTC), but its display defaults to the time zone of the server, the service config, the database, the user, or the session.
    - If you input a timestamp with no time zone and cast it to one with the time zone, PostgreSQL assumes the default time zone in effect. If you don't set your time zone in postgresql.conf, the server's default takes effect. This means that if you change your server's time zone, you'll see all the displayed times change after the PostgreSQL server restarts.
  - Timetz: It is time zone–aware but does not store the date.

Time Zones: What They Are and Are Not

- If you save 2022-11-03 05:32:00+02 (+2 being the Beirut offset from UTC), PostgreSQL internally thinks like this:
- Calculate the UTC time for 2022-11-03 05:32:00+02. This is 2022-11-03 03:32:00-0.
- Store the value 2022-11-03 03:32:00.
- So PostgreSQL doesn’t store the time zone, but uses it only to convert the datetime to UTC before storage.
- select date with timezone
  - select some_date at time zone 'Europe/Berlin' from some_table
  - SELECT '2012-02-28 10:00 PM America/Los_Angeles'::timestamptz AT TIME ZONE 'Europe/Paris';

Datetime Operators and Functions

- With intervals, we can add and subtract timestamp data simply by using the arithmetic operators we’re intimately familiar with.
  - The addition operator (+) adds an interval to a timestamp:
  - SELECT '2012-02-10 11:00 PM'::timestamp + interval '1 hour‘;

  - The subtraction operator (-) subtracts an interval from a temporal type:
  - SELECT '2012-02-10 11:00 PM'::timestamptz - interval '1 hour';

  - OVERLAPS, demonstrated in Example 5-12, returns true if two temporal ranges overlap.

```
SELECT
('2012-10-25 10:00 AM'::timestamp, '2012-10-25 2:00 PM'::timestamp)
OVERLAPS
('2012-10-25 11:00 AM'::timestamp,'2012-10-26 2:00 PM'::timestamp)
```

Arrays

Arrays

- PostgreSQL allows a table column to contain multi-dimensional arrays that can be of any built-in or user-defined data type.
- Arrays have an advantage over large plain text fields in that data remains in a discreet and addressable form.
- Every data type in Postgres has a companion array type.
- If you define your own data type, PostgreSQL creates a corresponding array type in the background for you.
  - For example, integer has an integer array type integer[], character has a character array type character[], …

Array Constructors

- The most rudimentary way to create an array is to type the elements:

```
SELECT ARRAY[2001, 2002, 2003] As yrs;
```

Create

- Assuming that you are going to create a table of people that contains a column called aliases that is a text array of various other names for a person.
- Creating an array column in PostgreSQL is as easy as appending the [ ] operator after your data type in a create statement such as this:

```
CREATE TABLE people
(   id serial,
   full_name text,
   aliases text[],
  CONSTRAINT people_id_pkey PRIMARY KEY (id) );
```

Insert

- Inserting data into a table with an array column can be as easy as:

  ```
  insert into people (full_name, aliases) values ('Abraham Lincoln', {"President Lincoln", "Honest Abe"}');
  ```

- Doing an insert into an array column requires using the **{ }** operator to specify the array and using double quotes (“) and a comma to delimit the separate values going into the array.

- We can see how the data went into our table by running:

  ```
  select * from people;
  ```

Access specific elements

- Our data are indeed separate inside of PostgreSQL looking at the *aliases* column.
  - You can access specific elements in the array by also using the **[ ]** notation in selects.

  ```
  select full_name, aliases[1] from people;
  update people set aliases[1] = 'President Abraham Lincoln' where id = 1;
  ```

- If you already know how many elements are in the array, you can add to it using standard syntax.

  ```
  update people set aliases[3] = 'President Lincoln' where id = 1;
  ```

- Note that array notation in PostgreSQL starts at element one and not zero

Access specific elements

- Using the array functions that PostgreSQL provides with varying numbers of array sizes.
- For example, if we want to append a value to an array we would use a query that makes use of the **array_append** function.

```
update people set aliases = array_append(aliases, 'Not Abe Vigoda') where id = 1;
```

- Using the array_remove function to remove an element.

```
update people set aliases = array_remove(aliases, 'Not Abe Vigoda') where id = 1;
```

-

Searching

- PostgreSQL provides the contains operators (@>, <@) for this type of search.
  - SELECT '{1,2,3}'::int[] @> '{3,2}'::int[] AS contains;
  - SELECT '{1,2,3}'::int[] <@ '{3,2}'::int[] AS contained_by;
  - select * from people where aliases @> '{"Prince Harry"}';

JSON

JSON

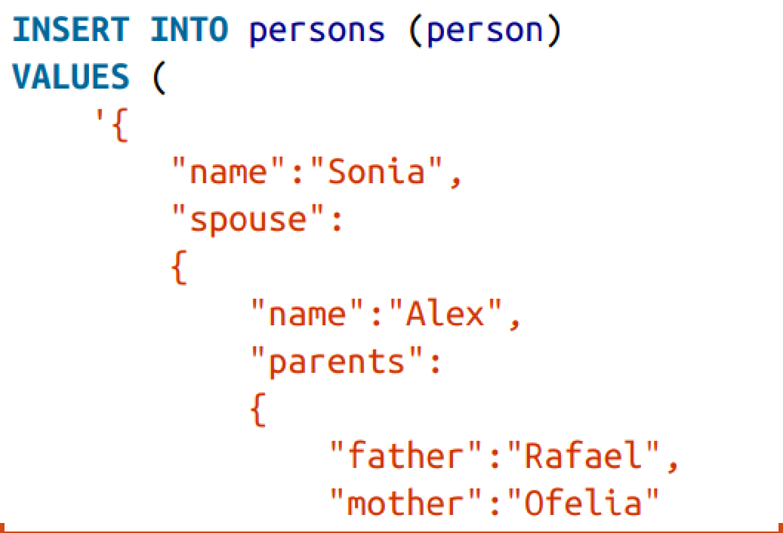
- PostgreSQL provides JSON (JavaScript Object Notation) and many support functions. JSON has become the most popular data interchange format for web applications.
- Version 9.3 significantly beefed up JSON support with new functions for extracting, editing, and casting to other data types.
- Version 9.4 introduced the JSONB data type, a binary form of JSON that can also take advantage of indexes.
- Version 9.5 introduced more functions for jsonb, including functions for setting elements in a jsonb object.
- Version 9.6 introduced the jsonb_insert function for inserting elements into an existing jsonb array or adding a new key value.

Inserting JSON Data

- To create a table to store JSON, define a column as a json type:

  CREATE TABLE persons (id serial PRIMARY KEY, person json);

- PostgreSQL automatically validates the input to make sure what you are adding is valid JSON.



```
INSERT INTO persons (person)
VALUES (
    '{
        "name":"Sonia",
        "spouse":
        {
            "name":"Alex",
            "parents":
            {
                "father":"Rafael",
                "mother":"Ofelia"
```

Querying JSON

- The easiest way to traverse the hierarchy of a JSON object is by using pointer symbols.
- SELECT person->'name' FROM persons;
- SELECT person->'spouse'->'parents'->'father' FROM persons;

- You can also write the query using a path array as in the following example:
- SELECT person#>array['spouse','parents','father'] FROM persons;

Querying JSON

- To penetrate JSON arrays, specify the array index. JSON arrays is zero-indexed, unlike PostgreSQL arrays, whose indexes start at 1.
- SELECT person->'children'->0->'name' FROM persons;
- And the path array equivalent:
- SELECT person#>array['children','0','name'] FROM persons;

Querying JSON

- PostgreSQL provides two native operators -> and ->> to help you query JSON data. The operator -> returns JSON object field as JSON. The operator ->> **returns JSON object field as text**.
- All queries in the prior examples return the value as JSON primitives (numbers,strings, booleans).
- To return the text representation, add another greater-than sign as in the following examples:
  - SELECT person->'spouse'->'parents'->>'father' FROM persons;
  - SELECT person#>>array['children','0','name'] FROM persons;

Querying JSON

```
SELECT id, person->'children'
       FROM public.persons
   where person->'children'->1->>'name'  = 'Azaleah'
```

json_array_elements to expand JSON array

- The json_array_elements function takes a JSON array and returns each element of the array as a separate row.

- SELECT json_array_elements(person->'children')->>'name' As name FROM persons;

Binary JSON: jsonb

It is handled through the same operators as those for the json type, and similarly named functions, plus several additional ones.

jsonb performance is much better than json performance because jsonb doesn’t need to be reparsed during operations.

Basics of PostgreSQL's JSONB data type

- Creating a JSONB column
  - create table sales (
  -     id serial not null primary key,
  -     info jsonb not null
  -);
- Inserting a JSON document
  - insert into sales values (1, '{"name": "Alice", "paying": true, "tags": ["admin"]}');

Update jsonb

- Updating by inserting a whole document:

```
update sales set info = '{" name " : "Bob", "paying" : false, "tags" : []}';
```

- Updating by adding a key:
  - Use the || operator to concatenate existing data with new data. The operator will either update or insert the key to the existing document.

```
update sales set info = info || '{"country": "Canada"}';
```

- Update by removing a key:
  - Use the - operator to remove a key from the document.

```
update sales set info = info - 'country';
```

Jsonb operators

- In addition to the operators supported by json, jsonb has additional comparator operators such as equality (=), contains (@>), contained (<@), …
- JSONB Containment with @>
- select info from sales where info::jsonb @> '{"name":"Bob"}'