#PostgreSQL Advanced Data Types

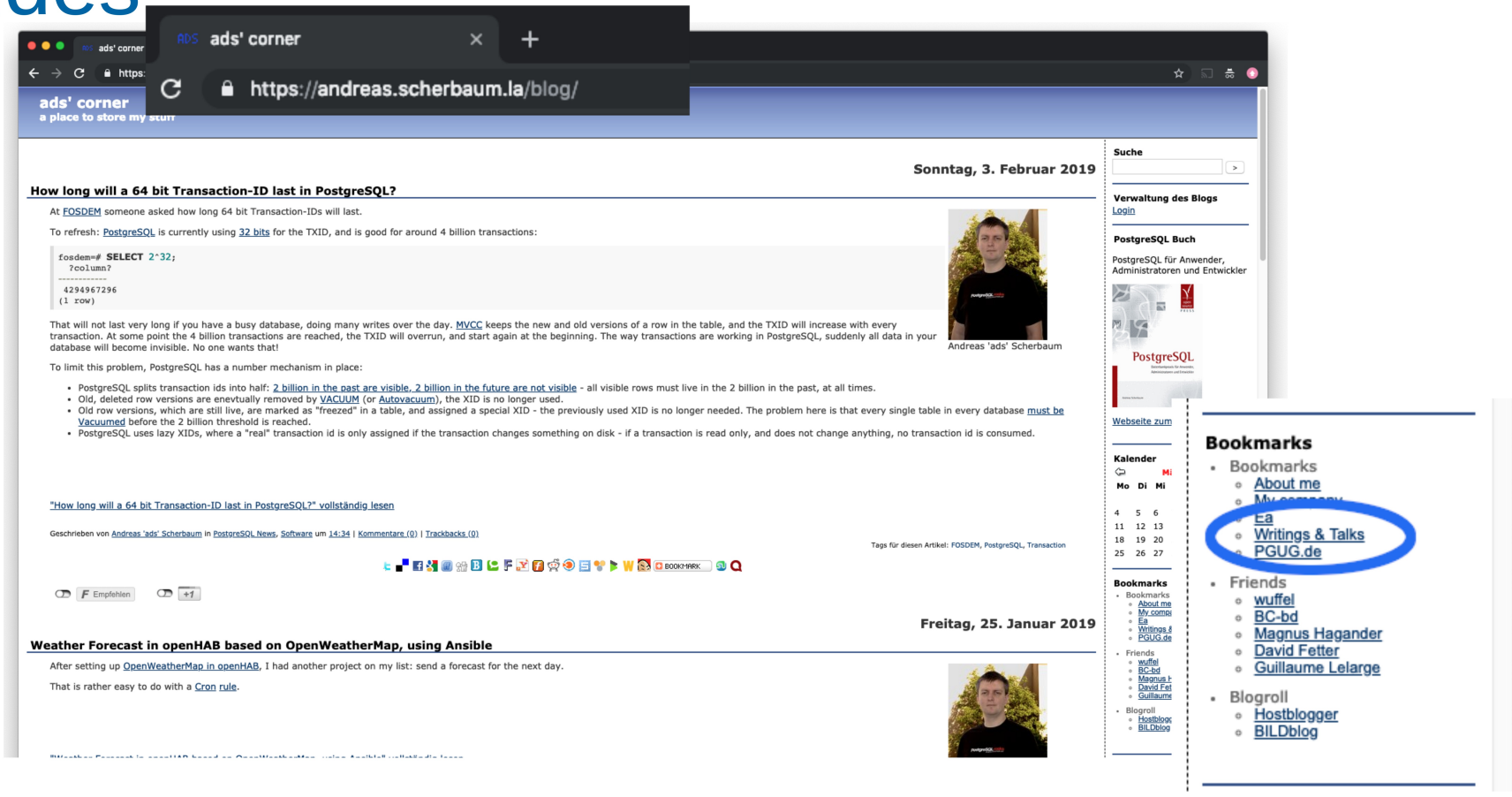
Andreas Scherbaum

PGConf.NYC 2021

Andreas Scherbaum

- Works with databases since ~1997, with PostgreSQL since ~1998
- Founding member of PGEU
- Board of Directors: PGEU, Orga team for pgconf.[eu|de], FOSDEM
- PostgreSQL Regional Contact for Germany
- Ran my own company around PostgreSQL for 7+ years
- Joined EMC in 2011
  - then Pivotal, then EMC, then Pivotal
  - working on PostgreSQL and Greenplum projects
- Joined Adjust GmbH in Berlin in 2020
  - Head of (around 1000) Databases

Slides


ads' corner
https://andreas.scherbaum.la/blog/
ads' corner
a place to store my stuff
Sonntag, 3. Februar 2019
How long will a 64 bit Transaction-ID last in PostgreSQL?
At FOSDEM someone asked how long 64 bit Transaction-IDs will last.
To refresh: PostgreSQL is currently using 32 bits for the TXID, and is good for around 4 billion transactions:
fosdem=# SELECT 2^32;
?column?
4294967296
(1 row)
That will not last very long if you have a busy database, doing many writes over the day. MVCC keeps the new and old versions of a row in the table, and the TXID will increase with every transaction. At some point the 4 billion transactions are reached, the TXID will overrun, and start again at the beginning. The way transactions are working in PostgreSQL, suddenly all data in your database will become invisible. No one wants that!
Andreas 'ads' Scherbaum
To limit this problem, PostgreSQL has a number mechanism in place:
PostgreSQL splits transaction ids into half: 2 billion in the past are visible, 2 billion in the future are not visible - all visible rows must live in the 2 billion in the past, at all times.
Old, deleted row versions are enevtually removed by VACUUM (or Autovacuum), the XID is no longer used.
Old row versions, which are still live, are marked as "freezed" in a table, and assigned a special XID - the previously used XID is no longer needed. The problem here is that every single table in every database must be Vacuumed before the 2 billion threshold is reached.
PostgreSQL uses lazy XIDs, where a "real" transaction id is only assigned if the transaction changes something on disk - if a transaction is read only, and does not change anything, no transaction id is consumed.
"How long will a 64 bit Transaction-ID last in PostgreSQL?" vollständig lesen
Geschrieben von Andreas 'ads' Scherbaum in PostgreSQL News, Software um 14:34 | Kommentare (0) | Trackbacks (0)
Tags für diesen Artikel: FOSDEM, PostgreSQL, Transaction
Empfehlen
+1
Freitag, 25. Januar 2019
Weather Forecast in openHAB based on OpenWeatherMap, using Ansible
After setting up OpenWeatherMap in openHAB, I had another project on my list: send a forecast for the next day.
That is rather easy to do with a Cron rule.
Suche
Verwaltung des Blogs
Login
PostgreSQL Buch
PostgreSQL für Anwender, Administratoren und Entwickler
Webseite zum
Kalender
Mo Di Mi
4 5 6
11 12 13
18 19 20
25 26 27
Bookmarks
Bookmarks
About me
Ea
Writings & Talks
PGUG.de
Friends
wuffel
BC-bd
Magnus Hagander
David Fetter
Guillaume Lelarge
Blogroll
Hostblogger
BILDblog

Get Social ...

- Twitter, Instagram: @ascherbaum
- LinkedIn: https://www.linkedin.com/in/andreas-scherbaum-87b6663/
- Mastodon: https://mastodon.social/@ascherbaum
- Blog: https://andreas.scherbaum.la/

Previous talk: Tour de Data Types

- Quick poll (1): how many of you have seen the previous talk?

Previous talk: Tour de Data Types

- Quick poll (2): how many of you do use more data types now?

Foreword

- Material for several days of training classes
- PostgreSQL is an ORDBMS (Object-relational database management system)
- Very extensible (own data types, extensions, FDW, …)
- This talk is focussed on what Data Types can do, rather than just listening more types

It’s all the same

- All data types in PostgreSQL are (technically) just an extension
- Even the basic ones

Agenda

- Arrays
- JSON(B)
- Composite Types
- DOMAINs
- ENUM Types
- Range Types
- Create your own type

Money

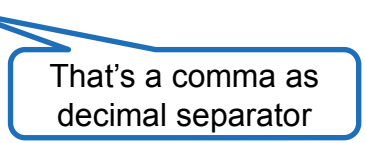
$$$

Money

- Holds only **one** currency
- No exchange rate possible
- Currency depends on system settings ($lc_monetary)
- Decimal separator is different per country
  - Sometimes even different in countries
- Precision is like NUMERIC

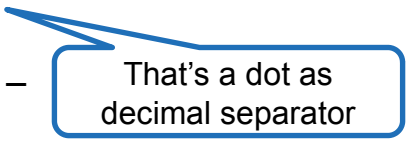
Money (Euro)

```
SHOW lc_monetary;
 lc_monetary
-------------
 de_DE.UTF-8
(1 row)


SELECT '123,45'::MONEY;
  money
----------
 123,45 €
(1 row)
```



Money (Euro)

```
SELECT '123.45'::MONEY;
    money
-------------
 12.345,00 €
(1 row)
```



Money (Dollar)

```
SET lc_monetary TO 'en_US.UTF-8';
SET

SELECT '123.45'::MONEY;
  money
---------
 $123.45
(1 row)
```

Money (Real)

```
SET lc_monetary TO 'pt_BR.UTF-8';
SET

SELECT '123.45'::MONEY;
    money
--------------
 R$ 12.345,00
(1 row)
```

Decimal separator

- Europe: majority of countries uses comma
  - Except: UK (dot), Switzerland and Liechtenstein (dot + ‘)
- USA + Canada: decimal point
  - Except certain parts of Canada (Francophone)
- South of US: mixed
- Asia: all mixed up
- Africa: no standard

Arrays

Arrays

- Every tuple can be a multi-dimensional array
- Can be any built-in, user-defined, Enum or composite type (no domains)
- Dimensions can be specified, but are ignored
- Creation by either using curly brackets, or ARRAY() constructor
- Very flexible (think: predecessor to JSON)

Use array in a table

```
CREATE TABLE addresses (
  …
  phone_number TEXT []
);

INSERT INTO addresses …
VALUES (…, ARRAY ['+1 555 1212', '+1 555 2368']);

INSERT INTO addresses …
VALUES (…, '{"+1 555 1212", "+1 555 2368"}');
```

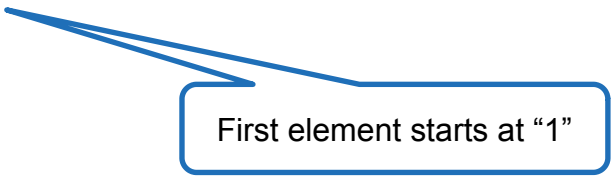
Define field as ARRAY

Use ARRAY constructor

Use curly braces, single + double quotes

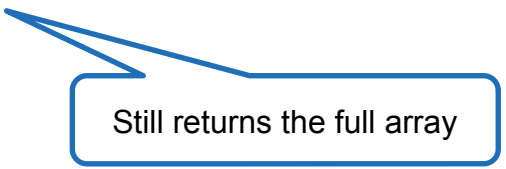
Select data from an array

```
SELECT phone_number FROM addresses;
          phone_number
--------------------------------
 {"+1 555 1212","+1 555 2368"}

SELECT phone_number[1] FROM addresses;
 phone_number
--------------
 +1 555 1212
```



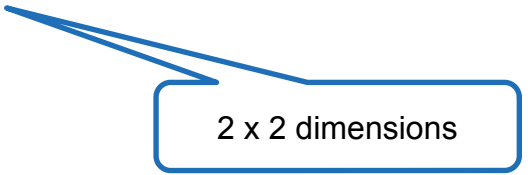
Find data in an array

```
SELECT …, phone_number FROM addresses
WHERE '+1 555 2368' = ANY(phone_number);
 … |           phone_number
---------------------------------------
 … | {"+1 555 1212","+1 555 2368"}
```



Multi-dimensional arrays

```
SELECT ARRAY[['abc', 'def'], ['123', '[456']];
          array
------------------------
 {{abc,def},{123,[456}}
```




```
SELECT array_dims(ARRAY[['abc', 'def'], ['123', '[456']]);
 array_dims
------------
 [1:2][1:2]
```

Select elements from array

```
SELECT (ARRAY['abc', 'def', 'ghi'])[1];
 array
-------
 abc
```

1st Element

```
SELECT (ARRAY['abc', 'def', 'ghi'])[2:3];
   array
-----------
 {def,ghi}
```

Element 2 to 3

Array dimension and length

```
SELECT array_dims(ARRAY[['abc', 'def'], ['123', '[456']]);
 array_dims
------------
 [1:2][1:2]
```

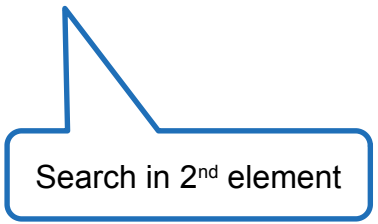
2 x 2 Dimensions

```
SELECT array_length(ARRAY[['abc', 'def'],['123', '[456']], 1);
 array_length
--------------
            2
```

2 elements in outer array

Search in array

```
SELECT TRUE WHERE (ARRAY['abc', 'def', 'ghi'])[2] = 'def';
 bool
------
 t
```



Modify array

```
SELECT array_prepend(0, ARRAY[1, 2, 3]);
 array_prepend
---------------
 {0,1,2,3}
(1 row)
```

Prepend this to the array

```
SELECT array_append(ARRAY[1, 2, 3], 4);
 array_append
--------------
 {1,2,3,4}
```

Append this to the array

Transform array

```
SELECT array_to_string(ARRAY['abc', 'def', 'ghi'], ' - ');
 array_to_string
-----------------
 abc - def - ghi
(1 row)

SELECT unnest(ARRAY['abc', 'def', 'ghi']);
 unnest
--------
 abc
 def
 ghi
```

Fill in between array elements

Transform array into rows

JSON(B)

JSON(B)

- PostgreSQL has 2 JSOB types
- JSON: stores the raw data, parses data “on access”
- JSONB: stores the parsed data, allows much greater functionality
- Generally: use JSONB over JSON
- Data manipulation is a function call, not a pure DML statement
  - jsonb Subscripting in v14

Create JSONB

```
CREATE TABLE addresses (
 …
  phone_number JSONB
);

INSERT INTO addresses …
VALUES (…, '["+1 555 1212", "+1 555 2368"]');
```

Define field as binary JSON

Inserts an array

Select from JSONB

```
SELECT phone_number FROM addresses;
          phone_number
--------------------------------
["+1 555 1212", "+1 555 2368"]

SELECT phone_number->1 FROM addresses;
 phone_number
---------------
 "+1 555 1212"
```

First element starts at “1”

Still a JSON object

Type of result

```
SELECT pg_typeof((
SELECT phone_number->1 FROM addresses));
 pg_typeof
-----------
   jsonb

SELECT (phone_number->1)::TEXT FROM addresses;
 phone_number
---------------
 "+1 555 1212"
```

Subquery

Now it's a PostgreSQL TEXT

Just JSONB

```
CREATE TABLE addresses (
  data  JSONB
);

INSERT INTO addresses(data)
VALUES ('{"name": "Slonik", "Birthdate": "4/3/1997"}');


INSERT INTO addresses(data)
VALUES ('{"name": "PGEU", "birthdate": "2008-03-21"}');
```

Go “all in” in NoSQL

Stored “as is”

Select NoSQL

```
SELECT COUNT(*) FROM addresses WHERE data ? 'Birthdate';
 count
-------
     1

SELECT (data->>'Birthdate')::DATE FROM addresses
WHERE data ? 'Birthdate';
    date
------------
 1997-04-03
```

Fix the mistake (rename a field)

```
UPDATE addresses
SET data = data - 'Birthdate' ||
```

Remove the “wrong” field

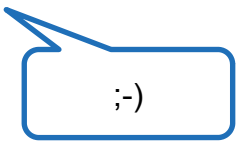
```
jsonb_build_object('birthdate', data->'Birthdate')
```

Add a new field
with the old value

```
WHERE data ? 'Birthdate';
```

Fix the mistake

```
SELECT * FROM addresses;
                      data
-----------------------------------------------
 {"name": "PGEU", "birthdate": "2008-03-21"}
 {"name": "Slonik", "birthdate": "4/3/1997"}
```



JSONB fields in queries

```
SELECT * FROM addresses
WHERE (data->>'birthdate')::DATE <
```

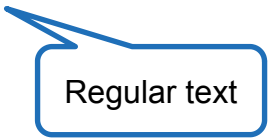
Conversation happens “on the fly”

```
(SELECT NOW()::DATE - INTERVAL '18 years');

                    data
-------------------------------------------
 {"name": "Slonik", "birthdate": "4/3/1997"}
```

“old” enough

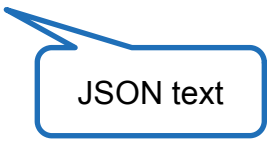
JSONB: Add a new field (version 1)

```
UPDATE addresses SET data = data ||
       jsonb_build_object('hobby', 'databases');

SELECT * FROM addresses;
                    data
-----------------------------------------
 {"name": "PGEU", "hobby": "databases",
  "birthdate":    "2008-03-21"}
 {"name": "Slonik", "hobby": "databases",
  "birthdate": "4/3/1997"}
```



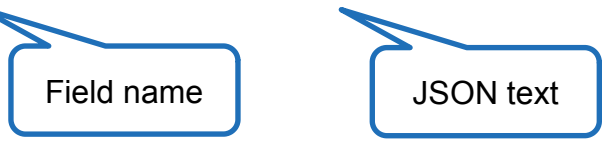
JSONB: Add a new field (version 2)

```
UPDATE addresses
SET data = jsonb_insert(data, '{hobby}', '"databases"');

SELECT * FROM addresses;
                     data
-----------------------------------------
 {"name": "PGEU", "hobby": "databases",
  "birthdate":    "2008-03-21"}
 {"name": "Slonik", "hobby": "databases",
  "birthdate": "4/3/1997"}
```



JSONB: update field

```
UPDATE addresses
SET data = jsonb_set(data, '{hobby}', '"PostgreSQL"');

SELECT * FROM addresses;
                    data
-----------------------------------------
 {"name": "PGEU", "hobby": "PostgreSQL",
  "birthdate": "2008-03-21"}
 {"name": "Slonik", "hobby": "PostgreSQL",
  "birthdate": "4/3/1997"}
```



Composite Types

Composite Types

- Also called: **Row Type**, or **Record**
- List of field names and according data types
- Can be used (almost) anywhere where a simple type can be used
- No constraints can be used
- Every table has an according row type with the same structure

Create a Composite Type

```
CREATE TYPE currency_value AS (
  cv_currency          CHAR(3),
  cv_other_currency    CHAR(3),
  cv_date              DATE,
  cv_value             NUMERIC(10,3)
);

CREATE TABLE currency_history (
  id                   SERIAL    PRIMARY KEY,
  data                 currency_value
);
```

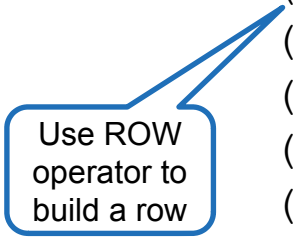
AS keyword is mandatory

Looks like a table

Used like any other data type

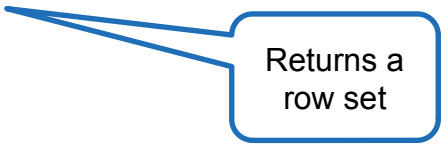
Use a Composite Type

```
INSERT INTO currency_history (data)
     VALUES (ROW('EUR', 'USD', '2021-01-31', 1.0755)),
            (ROW('EUR', 'USD', '2021-01-30', 1.0630)),
            (ROW('EUR', 'USD', '2021-01-27', 1.0681)),
            (ROW('EUR', 'USD', '2021-01-26', 1.0700)),
            (ROW('EUR', 'USD', '2021-01-25', 1.0743)),
            (ROW('EUR', 'USD', '2021-01-24', 1.0748)),
            (ROW('EUR', 'USD', '2021-01-23', 1.0715));
```



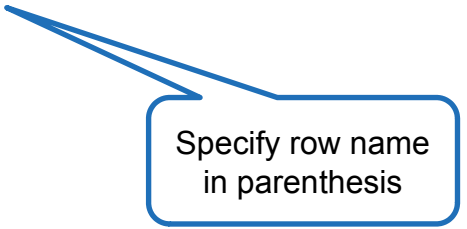
Result set, including a Composite Type

```
SELECT * FROM currency_history;
 id |              data
----+--------------------------------
  1 | (EUR,USD,2021-01-31,1.076)
  2 | (EUR,USD,2021-01-30,1.063)
  3 | (EUR,USD,2021-01-27,1.068)
  4 | (EUR,USD,2021-01-26,1.070)
  5 | (EUR,USD,2021-01-25,1.074)
  6 | (EUR,USD,2021-01-24,1.075)
  7 | (EUR,USD,2021-01-23,1.072)
(7 rows)
```



Select specific fields from a Composite Type

```
SELECT (data).cv_date, (data).cv_value FROM currency_history;
  cv_date    | cv_value
-------------+----------
 2021-01-31 |    1.076
 2021-01-30 |    1.063
 2021-01-27 |    1.068
 2021-01-26 |    1.070
 2021-01-25 |    1.074
 2021-01-24 |    1.075
 2021-01-23 |    1.072
(7 rows)
```



DOMAINs

DOMAINs

- Single data type, with optional constraints (restrictions) attached
- SQL Standard feature
- NOT NULL / NULL can be added
- DEFAULT value can be added (as example: nextval(‘sequence’))
- CHECK constraints can be added
- Underrated feature in PostgreSQL

Use Cases

- Apply basic integrity checks
- Verify that units are specified
- Verify that values are in a certain range (age, height, prices, dates)
- Verify that dimensions are specified, and valid (m/s, km/h, MB/s)
- Verify that an SRID is specified (coordinates)

- All of this can be done on the table/column level
- Adding it to the data type saves work when used multiple times

Create a DOMAIN

```
CREATE DOMAIN md5_sum AS TEXT
NOT NULL
CHECK (
  LENGTH(VALUE) = 32
);

CREATE TABLE checksums (
  filename  TEXT PRIMARY KEY,
  checksum  md5_sum
);
```

Use a DOMAIN

```
INSERT INTO checksums VALUES
('passwd', 'abc123def');
ERROR:  value for domain md5_sum violates check constraint
"md5_sum_check”


INSERT INTO checksums VALUES
('passwd', '5e1ffef4dcdaf4a011b57fd9d525a71f');
INSERT 0 1
```

Change a DOMAIN

- SET/DROP NOT NULL
- ADD/DROP/VALIDATE CONSTRAINT

- ADD will check all columns, except if “NOT VALID” is used
- Newly added values in tables will always be checked
- Can be validated later using VALIDATE

- OWNER TO
- RENAME TO
- SET SCHEMA

Enumeration Type (ENUM)

ENUMs

- List of allowed values in a tuple
- Replaces a lookup table (1:n table)
- Values are case-sensitive

- A DOMAIN can also be used, with a CHECK constraint for allowed values
- A domain can use UPPER()/LOWER() to make values case-insensitive

Create an ENUM (and a DOMAIN)

```
CREATE TYPE traffic_light1 AS ENUM ('red', 'yellow', 'green');

CREATE DOMAIN traffic_light2 AS TEXT NOT NULL
CHECK(
  LOWER(VALUE) IN ('red', 'yellow', 'green')
);
```

Create a table with the new types

```
CREATE TABLE traffic_data (
  light1    traffic_light1 NOT NULL,
  light2    traffic_light2
);
```

INSERT data

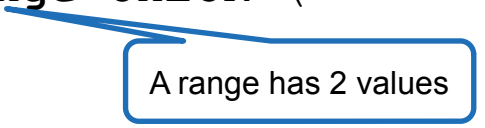
```
INSERT INTO traffic_data VALUES ('red', 'red');
INSERT 0 1

INSERT INTO traffic_data VALUES ('Red', 'red');
ERROR:  invalid input value for enum traffic_light1: "Red"
LINE 1: INSERT INTO traffic_data VALUES ('Red', 'red');
                                         ^

postgres=# INSERT INTO traffic_data VALUES ('red', 'Red');
INSERT 0 1
```

SELECT data

```
SELECT * FROM traffic_data;

 light1 | light2
--------+--------
 red    | red
 red    | Red
```

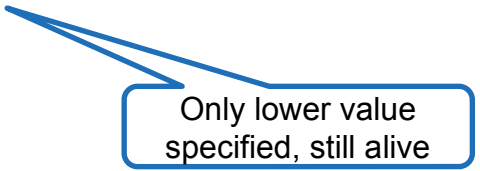
Human lifetime check

```
CREATE DOMAIN lifespan AS daterange CHECK (
   (NOT lower_inf(VALUE)) AND (
    upper_inf(VALUE) OR
    (upper(VALUE) - lower(VALUE) < 365*150)
   )
);
```



Verify age based on birth

```
SELECT '[2021-09-12,)'::lifespan;
   lifespan
---------------
 [2021-09-12,)
```



Death before birth?

```
SELECT '[2021-09-12,1900-01-01)'::lifespan;
ERROR:  range lower bound must be less than or equal to range
upper bound
LINE 1: SELECT '[2021-09-12,1900-01-01)'::lifespan;
               ^
```

Birthdate and death are in range

```
SELECT '[1900-01-01,2021-09-12)'::lifespan;
        lifespan
-------------------------
 [1900-01-01,2021-09-12)
```

Date out of range: too old

```
SELECT '[1800-01-01,2021-09-12)'::lifespan;
ERROR:  value for domain lifespan violates check constraint
"lifespan_check"
```

Change an ENUM

- ALTER TYPE … ADD VALUE ‘black‘ AFTER ‘green‘;
- ALTER TYPE ... RENAME VALUE ‘black‘ TO ‘white‘;

- ALTER TYPE ... DROP VALUE ?

- OWNER TO
- RENAME TO
- SET SCHEMA

Range Types

Range Types

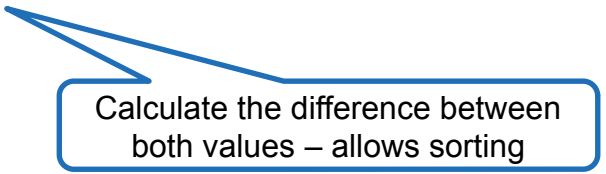
- Range Types represent a “**range of values**”, of a certain “**subtype**” (basic date type)

- Date range: range of dates (daterange)
- Time range: range of times (tsrange, tstzrange)
- Integer range: range of integers (int4range, int8range)
- Numeric range: range of numerics (numrange)

- Create your own!

Range Types

```
CREATE TYPE float8range AS RANGE (
            subtype = float8,
            subtype_diff = float8mi);
```




```
SELECT '[23.12, 42.12]'::float8range;
  float8range
---------------
 [23.12,42.12]
```

Use the new Range Type

```
CREATE TABLE ranges8 (
  range8   float8range
);

CREATE INDEX ON ranges8 USING GIST(range8);

INSERT INTO ranges8 VALUES
('[1.0, 5.0]') , ('[3.0, 7.0]'), ('[2.0, 6.0]');
```

Use the new Range Type

```
SELECT * FROM ranges8 ORDER BY range8 DESC;
 range8
--------
 [3,7]
 [2,6]
 [1,5]
```

Use the new Range Type

```
SET enable_seqscan TO off;
```

Force planner to use available index

```
EXPLAIN SELECT * FROM ranges8 ORDER BY range8 DESC;
                            QUERY PLAN
------------------------------------------------------------------- Sort
 (cost=8.20..8.21 rows=3 width=32)
   Sort Key: range8 DESC
   ->  Index Only Scan using ranges8_range8_idx on ranges8
       (cost=0.13..8.18 rows=3 width=32)
(3 rows)
```

Constraints on Range Types

- Range Types allow different types of constraints
- EXCLUDE can be used to verify that two compared rows don’t return TRUE on all comparisations

- Prime example: book a hotel room

Book a hotel room

```
CREATE EXTENSION btree_gist;
```

Extension required for GIST =

```
CREATE TABLE hotel_reservation (
    stay DATERANGE,
    room_number TEXT,
    EXCLUDE USING GIST (room_number WITH =, stay WITH &&)
);
```

Either rooms can be equal

Or stay range can overlap

Book a hotel room

```
INSERT INTO hotel_reservation VALUES
(DATERANGE(DATE '2019-09-11', '2019-09-13'), '42');

INSERT INTO hotel_reservation VALUES
(DATERANGE(DATE '2019-09-12', '2019-09-13'), '23');
```

Double-book a room

```
INSERT INTO hotel_reservation VALUES
(DATERANGE(DATE '2019-09-12', '2019-09-13'), '42');
```

```
ERROR:  conflicting key value violates exclusion constraint
"hotel_reservation_room_number_stay_excl"
DETAIL:  Key (room_number, stay)=(42, [2019-09-12,2019-09-13))
conflicts with existing key (room_number, stay)=(42, [2019-09-
11,2019-09-13)).
```

Constraint operations on Range Types

Depending on the type, many different comparisation options are possible

equal (=), **not equal** (<>), **less than** (<), **greater than** (>), less than and equal (<=), greater than and equal (>=), contains range/element (@>), range/element is contained by (<@), **overlap** (&&), strictly left (<<), strictly right (>>), adjacent to (-|-), union (+), intersection (*), difference (-)

Access range boundaries

```
SELECT LOWER(DATERANGE(DATE '2019-09-11', '2019-09-13'));
   lower
------------
 2019-09-11

SELECT UPPER(DATERANGE(DATE '2019-09-11', '2019-09-13'));
   upper
------------
 2019-09-13
```

Check for non-existent values

```
SELECT LOWER_INF(DATERANGE(DATE '2019-09-12', NULL));
 lower_inf
-----------
 f
```

Lower bound is specified

```
SELECT UPPER_INF(DATERANGE(DATE '2019-09-12', NULL));
 upper_inf
-----------
 t
```

Upper bound is not specified

Create your own type

Create your own type

- PostgreSQL allows you to create your very own data types
- Disclaimer: the following part is deep technical

- Even if all the existing types and extensions are not enough, PostgreSQL provides you the ability to extend the database even further!

Poll

- How does (the native) PHP PostgreSQL driver handle booleans?

- That was before PDO and other drivers …

PostgreSQL Boolean in native PHP

```
# PostgreSQL: t/f (boolean)
$boolean = pg_fetch_row($result)[0];

if ($boolean) {
  print "true";
} else {
  print "false";
}
```

Answer

- It doesn’t
- The native driver does not handle the boolean type, it’s always a string

PostgreSQL Boolean in native PHP

```
# PostgreSQL: t/f (boolean)
$boolean = pg_fetch_row($result)[0];

if ($boolean) {
  print "true";
} else {
  print "false";
}
```

Fetches the boolean value as a string

```
true
```

Is always “true”

Problem description

- Every check for boolean must always check for “t” and “f”
- Never use PHP native true/false
- That just calls for errors and mistakes

Workaround

- Return **0**/**1** instead of “**t**” and “**f**” – use a different data type in PostgreSQL
- Things are easy: input and output for “integers” already exist
- No new functions must be written

- Still plenty of new supporting functions, operators, and operator classes required
- We want the new type to be compatible with the existing boolean type

CREATE TYPE

- [0]: you must be superuser
- Input and Output function are required, other functions are optional
- **Input** function: converts textual representation into internal representation
- **Output** function: converts internal representation into external representation
- **Receive** function: external binary representation into internal
- **Send** function: internal representation into external binary

CREATE TYPE

- Type modifier input/output function
- Analyze function: overrides attempts to use “equal” and “less-than” functions

PostgreSQL Boolean in native PHP

```
CREATE FUNCTION boolean2_in(cstring)
   RETURNS boolean2
   AS 'boolin'
   LANGUAGE internal STRICT;

CREATE FUNCTION boolean2_out(boolean2)
   RETURNS cstring
   AS 'int2out'
   LANGUAGE internal STRICT;
```

Use existing function

Input is “cstring”

Use existing function

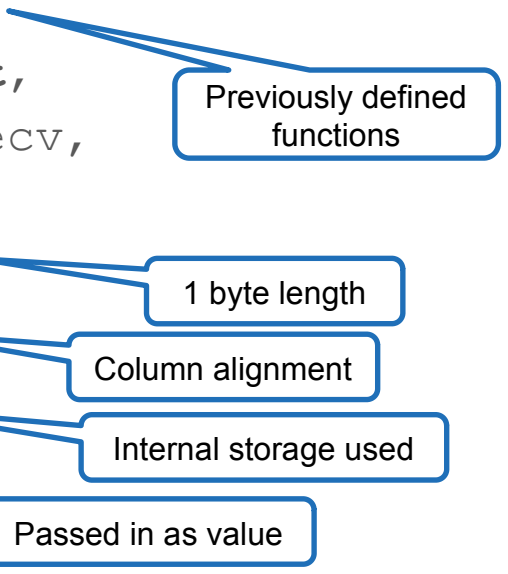
Output is the target type

PostgreSQL Boolean in native PHP

```
CREATE FUNCTION boolean2_recv(internal)
   RETURNS boolean2
   AS 'boolrecv'
   LANGUAGE internal STRICT;

CREATE FUNCTION boolean2_send(boolean2)
   RETURNS bytea
   AS 'boolsend'
   LANGUAGE internal STRICT;
```

PostgreSQL Boolean in native PHP

```
CREATE TYPE boolean2 (
    input = boolean2_in,
    output = boolean2_out,
    receive = boolean2_recv,
    send = boolean2_send,
    internallength = 1,
    alignment = char,
    storage = plain,
    passedbyvalue
);

COMMENT ON TYPE boolean2 IS 'boolean, ''1''/''0''';
```



Type created

- Bare minimum to make the type work
- Operators and supporting functions required as well
- First for “less than” between the original boolean type, and the new boolean type

Supporting functions

- For completeness, the following slides show all necessary supporting functions
- We define every aspect of how PostgreSQL handles data types internally

PostgreSQL Boolean in native PHP

```
CREATE FUNCTION boollt(boolean2, boolean)
   RETURNS boolean
   AS 'boollt'
   LANGUAGE internal STRICT;
```

Lower then

“old” versus “new”

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR < (
    PROCEDURE = boollt,
    LEFTARG = boolean2,
    RIGHTARG = boolean,
    COMMUTATOR = >,
    NEGATOR = >=,
    RESTRICT = scalarltsel,
    JOIN = scalarltjoinsel
);
```

For previously defined function

How to compare both values

How to join both values

PostgreSQL Boolean in native PHP

```
CREATE FUNCTION boollt(boolean, boolean2)
   RETURNS boolean
   AS 'boollt'
   LANGUAGE internal STRICT;
```

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR < (
    PROCEDURE = boollt,
    LEFTARG = boolean,
    RIGHTARG = boolean2,
    COMMUTATOR = >,
    NEGATOR = >=,
    RESTRICT = scalarltsel,
    JOIN = scalarltjoinsel
);
```

PostgreSQL Boolean in native PHP

```
CREATE FUNCTION boollt(boolean2, boolean2)
   RETURNS boolean
   AS 'boollt'
   LANGUAGE internal STRICT;
```

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR < (
    PROCEDURE = boollt,
    LEFTARG = boolean2,
    RIGHTARG = boolean2,
    COMMUTATOR = >,
    NEGATOR = >=,
    RESTRICT = scalarltsel,
    JOIN = scalarltjoinsel
);
```

Type created

- Then for “less or equal” between the original and new boolean type
- And “not equal”
- And “equal”
- And “greater then”
- And “greater equal”

PostgreSQL Boolean in native PHP

```
CREATE FUNCTION boolle(boolean2, boolean)
   RETURNS boolean
   AS 'boolle'
   LANGUAGE internal STRICT;
CREATE FUNCTION boolne(boolean2, boolean)
   RETURNS boolean
   AS 'boolne'
   LANGUAGE internal STRICT;
CREATE FUNCTION booleq(boolean2, boolean)
   RETURNS boolean
   AS 'booleq'
   LANGUAGE internal STRICT;
```

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR <= (
    PROCEDURE = boolle,
    LEFTARG = boolean2,
    RIGHTARG = boolean,
    COMMUTATOR = >=,
    NEGATOR = >,
    RESTRICT = scalarltsel,
    JOIN = scalarltjoinsel
);
```

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR <= (
    PROCEDURE = boolle,
    LEFTARG = boolean2,
    RIGHTARG = boolean,
    COMMUTATOR = >=,
    NEGATOR = >,
    RESTRICT = scalarltsel,
    JOIN = scalarltjoinsel
);
```

Supporting functions created

- Many functions (20) and operators (20) later …
- We are able to compare native and „PHP“ boolean values
- But index support (operator classes) is still missing!

PostgreSQL Boolean in native PHP

```
CREATE FUNCTION btboolcmp(boolean2, boolean)
   RETURNS int4
   AS 'btboolcmp'
   LANGUAGE internal STRICT;
CREATE FUNCTION btboolcmp(boolean, boolean2)
   RETURNS int4
   AS 'btboolcmp'
   LANGUAGE internal STRICT;
CREATE FUNCTION btboolcmp(boolean2, boolean2)
   RETURNS int4
   AS 'btboolcmp'
   LANGUAGE internal STRICT;
```

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR CLASS _bool2_ops
    DEFAULT FOR TYPE boolean2[] USING gin AS
    STORAGE boolean2 ,
    OPERATOR 1 &&(anyarray,anyarray) ,
    OPERATOR 2 @>(anyarray,anyarray) ,
    OPERATOR 3 <@(anyarray,anyarray) RECHECK ,
    OPERATOR 4 =(anyarray,anyarray) RECHECK ,
    FUNCTION 1 btboolcmp(boolean2,boolean2) ,
    FUNCTION 2 ginarrayextract(anyarray,internal) ,
    FUNCTION 3 ginarrayextract(anyarray,internal) ,
    FUNCTION 4 ginarrayconsistent(internal,smallint,internal);
```

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR CLASS bool2_ops
    DEFAULT FOR TYPE boolean2 USING btree AS
    OPERATOR 1 <(boolean2,boolean2) ,
    OPERATOR 2 <=(boolean2,boolean2) ,
    OPERATOR 3 =(boolean2,boolean2) ,
    OPERATOR 4 >=(boolean2,boolean2) ,
    OPERATOR 5 >(boolean2,boolean2) ,
    FUNCTION 1 btboolcmp(boolean2,boolean2);
```

PostgreSQL Boolean in native PHP

```
CREATE OPERATOR CLASS bool2_ops
    DEFAULT FOR TYPE boolean2 USING hash AS
    OPERATOR 1 =(boolean2,boolean2) ,
    FUNCTION 1 hashchar("char");
```

Summary

- Create input/output/receive/send functions
- Create base type
- Create cast functions and casts for other compatible types
- Create operators and supporting functions
- Create operator classes and supporting functions
- Depending on your use case, you might use existing functions, or have to write new ones

Test case

```
CREATE TABLE boolean2_test (
  id      SERIAL            NOT NULL PRIMARY KEY,
  test    boolean2
);

CREATE INDEX t2_test ON boolean2_test2(test);

INSERT INTO boolean2_test (test) VALUES ('true');
INSERT INTO boolean2_test (test) VALUES ('false');
```

Test case

```
SELECT id, test FROM boolean2_test ORDER BY id;
id | test
-----------
 1  |  1
 2  |  0
```

Questions?

The End