Tour de (PostgreSQL) Data Types

Andreas Scherbaum
PGConf Brasil 2019
Tour de (PostgreSQL) Data Types

Andreas Scherbaum

Picture: SCHEMA ELEPHANTI
Author: Jean Boch (Belgian, 1545-1608)
Date: 1595
Book: Descriptio pbylicae gratulationis, spectacvlorvm et lvdorvm, in aeventv sereniss: Principis Ernesti Archidvcis Avstriae Dvcis Vrgvndiae
Source: Metropolitan Museum of Art
Accession Number: 239 B63 Q
License: Public Domain
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
How long will a 64 bit Transaction-ID last in PostgreSQL?

At PostgreSQL someone asked how long 64 bit Transaction-IDs will last.
To refresh: PostgreSQL is currently using 32 bit for the TXID, and is good for around 4 billion transactions:

```
EXPLAIN SELECT 2**32;
```
```
4294967296
```
That will not last very long if you have a busy database, doing many writes over the day. PostgreSQL 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 overflow, and start again at the beginning. The way transactions are working in PostgreSQL, suddenly all data in your database will become invalid. No one wants that!

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 eventually removed by Vакuum or Vакuumтор, the TXID is no longer used.
- Old row versions, which are still live, are marked as "frozen" 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.

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 "Environment" rule.
What is PostgreSQL?

- (Most advanced) (Open Source) relational database, under BSD license.
- World-wide active community
- Many features, like Foreign Keys, Transactions, Trigger
- Runs on many platforms
- Aims for SQL standard compatibility
- About one major release per year
- 11 is the current version, next is 12
Behavioral Notes

• No WhatsApp/Twitter/… sounds, please
• Sleep only from row 5 (no snoring, please)
• Computer games and coding only from row 10 (please avoid rapid key clicks)
• Throwing blue Elephant stressballs is OK
Data Types in PostgreSQL

Quick poll (1): how many data types in PostgreSQL?
Data Types in PostgreSQL

SELECT COUNT(*) AS "Number Data Types"
FROM pg_catalog.pg_type;

Number Data Types
-------------------------------
361
Data Types in PostgreSQL

SELECT COUNT(*) AS "Number Data Types"
FROM pg_catalog.pg_type
WHERE typelem = 0
AND typrelid = 0;

Number Data Types
--------------------------------
82

> 0 references another type
> 0 references pg_class (table types)
**Data Types in PostgreSQL**

```sql
SELECT STRING_AGG(typname, '  ') AS "Data Types"
  FROM pg_catalog.pg_type
WHERE typelem = 0
  AND typrelid = 0;
```

**Data Types**

```
bool bytea char int8 int2 int4 regproc text oid tid xid cid json xml pg_node_tree smgr path polygon
float4 float8 abstime reltime tinterval unknown circle money macaddr inet cidr aclitem bpchar varchar
date time timestamp timestampz interval timetz bit varbit numeric refcursor regprocedure regoper
regoperator regclass regtype uuid pg_lsn tsvector gtsvector tsquery regconfig regdictionary jsonb
txid_snapshot int4range numrange tsrange tstzrange daterange int8range record cstring any anyarray
void trigger event_trigger language_handler internal opaque anyelement anynonarray anyenum
fdw_handler anyrange cardinal_number character_data sql_identifier time_stamp yes_or_no
```
# Data Types in PostgreSQL

<table>
<thead>
<tr>
<th>Name</th>
<th>Aliases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigint</td>
<td>int8</td>
<td>signed eight-byte integer</td>
</tr>
<tr>
<td>bigserial</td>
<td>serial0</td>
<td>autoncrementing eight-byte integer</td>
</tr>
<tr>
<td>bit [ (n) ]</td>
<td>varbit</td>
<td>variable-length bit string</td>
</tr>
<tr>
<td>boolean</td>
<td>bool</td>
<td>logical Boolean (true/false)</td>
</tr>
<tr>
<td>box</td>
<td></td>
<td>rectangular box on a plane</td>
</tr>
<tr>
<td>bytea</td>
<td></td>
<td>binary data (byte array)</td>
</tr>
<tr>
<td>character [ (n) ]</td>
<td>char [ (n) ]</td>
<td>fixed-length character string</td>
</tr>
<tr>
<td>character varying [ (n) ]</td>
<td>varchar [ (n) ]</td>
<td>variable-length character string</td>
</tr>
<tr>
<td>cidr</td>
<td></td>
<td>IPv4 or IPv6 network address</td>
</tr>
<tr>
<td>circle</td>
<td></td>
<td>circle on a plane</td>
</tr>
<tr>
<td>date</td>
<td></td>
<td>calendar data (year, month, day)</td>
</tr>
<tr>
<td>double precision</td>
<td>float8</td>
<td>double precision floating-point number (8 bytes)</td>
</tr>
<tr>
<td>inet</td>
<td></td>
<td>IPv4 or IPv6 host address</td>
</tr>
<tr>
<td>integer</td>
<td>int, int4</td>
<td>signed four-byte integer</td>
</tr>
<tr>
<td>interval [ fields ] [ (p) ]</td>
<td>time span</td>
<td>time span</td>
</tr>
<tr>
<td>json</td>
<td></td>
<td>textual JSON data</td>
</tr>
<tr>
<td>jsonb</td>
<td></td>
<td>binary JSON data, decomposed</td>
</tr>
<tr>
<td>line</td>
<td></td>
<td>infinite line on a plane</td>
</tr>
<tr>
<td>tsseg</td>
<td></td>
<td>line segment on a plane</td>
</tr>
<tr>
<td>macaddr</td>
<td></td>
<td>MAC (Media Access Control) address</td>
</tr>
<tr>
<td>money</td>
<td></td>
<td>currency amount</td>
</tr>
<tr>
<td>numeric [ (p, s) ]</td>
<td>decimal [ (p, s) ]</td>
<td>exact numeric of selectable precision</td>
</tr>
<tr>
<td>path</td>
<td></td>
<td>geometric path on a plane</td>
</tr>
<tr>
<td>pg_lsn</td>
<td></td>
<td>PostgreSQL Log Sequence Number</td>
</tr>
<tr>
<td>point</td>
<td></td>
<td>geometric point on a plane</td>
</tr>
<tr>
<td>polygon</td>
<td></td>
<td>closed geometric path on a plane</td>
</tr>
<tr>
<td>real</td>
<td>float4</td>
<td>single precision floating-point number (4 bytes)</td>
</tr>
<tr>
<td>smallint</td>
<td>int2</td>
<td>signed two-byte integer</td>
</tr>
<tr>
<td>smallserial</td>
<td>serial2</td>
<td>autoncrementing two-byte integer</td>
</tr>
<tr>
<td>serial</td>
<td>serial4</td>
<td>autoncrementing four-byte integer</td>
</tr>
<tr>
<td>text</td>
<td></td>
<td>variable-length character string</td>
</tr>
<tr>
<td>time [ (p) ] [ without time zone ]</td>
<td></td>
<td>time of day (no time zone)</td>
</tr>
<tr>
<td>time [ (p) ] with time zone</td>
<td>timez</td>
<td>time of day, including time zone</td>
</tr>
<tr>
<td>timestamp [ (p) ] [ without time zone ]</td>
<td></td>
<td>date and time (no time zone)</td>
</tr>
<tr>
<td>timestamp [ (p) ] with time zone</td>
<td>timestampz</td>
<td>date and time, including time zone</td>
</tr>
<tr>
<td>textsearch</td>
<td></td>
<td>text search query</td>
</tr>
<tr>
<td>tsvector</td>
<td></td>
<td>text search document</td>
</tr>
<tr>
<td>txid_snapshot</td>
<td></td>
<td>user-local transaction ID snapshot</td>
</tr>
<tr>
<td>uuid</td>
<td></td>
<td>universally unique identifier</td>
</tr>
<tr>
<td>xml</td>
<td></td>
<td>XML data</td>
</tr>
</tbody>
</table>
Data Types in PostgreSQL

Another poll (2): how many different data types are you using?
Agenda

- Text Types
- Numeric Types
- Dates and Times (Temporal Types)
- XML
- JSON
- Boolean
- Bits
- Binary Data
- Network
- Create your own Data Type
Agenda

- Text Types
- Numeric Types
- Dates and Times
- XML
- JSON
- Boolean
- Bits
- Binary Data
- Network
- Create your own Data Type
Text Types

- VARCHAR (optional: length)
- CHAR (optional: length)
- TEXT

- Internally: it’s the same
- Note: text types are case sensitive
Text Types

- VARCHAR: string up to ~1GB
- VARCHAR(n): string up to length ‘n’, except trailing whitespaces
- CHAR: 1 byte string
- CHAR(n): string with length ‘n’
- TEXT: string up to ~1GB
Text Types: VARCHAR versus CHAR

SELECT octet_length('abcde'::VARCHAR(1)) as "vc_1",
octet_length('abcde'::VARCHAR(5)) as "vc_5",
octet_length('abcde'::VARCHAR(10)) as "vc_10",
octet_length('abcde'::CHAR(1)) as "c_1",
octet_length('abcde'::CHAR(5)) as "c_5",
octet_length('abcde'::CHAR(10)) as "c_10";

vc_1   |   vc_5   |   vc_10   |   c_1   |   c_5   |   c_10   
-------+----------+-----------+---------+---------+---------
     1   |       5   |       5   |       1   |       5   |      10   
(1 row)

LENGTH() and CHAR_LENGTH() return '0'
You don’t want CHAR(n)

- CHAR(n) will be padded with white spaces, and displayed that way
- Trailing spaces are insignificant (usually, when char characters are compared)
- Whitespaces are significant in certain collations
- Whitespaces are significant in VARCHAR and TEXT
- Space padding wastes space
- It’s not a fixed-width type, and the storage space depends on the characters
- Shorter values are not rejected, just padded with space
Be careful with VARCHAR(n)

- The length limit is just the maximum length, not an enforcement
- Use VARCHAR or TEXT instead, add a CHECK constraint
Internals: Pages & TOAST

BLCKSZ = 8192  
(src/include/pg_config.h)

TOAST_TUPLES_PER_PAGE = 4  
TOAST_TUPLE_THRESHOLD = 2032  
TOAST_TUPLE_TARGET = 2032  
(src/include/access/tuptoaster.h)
## Internals: Pages & TOAST

<table>
<thead>
<tr>
<th>Row Header</th>
<th>TOAST Page Header</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INT</strong></td>
<td><strong>Row</strong></td>
</tr>
<tr>
<td><strong>TEXT</strong></td>
<td><strong>Row</strong></td>
</tr>
<tr>
<td><strong>INT</strong></td>
<td></td>
</tr>
</tbody>
</table>

- Page 8 kB
- TOAST Page 8 kB
- 4 Byte Pointer
What about CHAR(255)?

```
varchar(255) sql
```

About 430,000 results (0.59 seconds)

```
char(255) sql
```

About 411,000 results (0.45 seconds)
What about CHAR(255)?

• Does not apply to PostgreSQL
• Probably arbitrary choice: $255 = 2^8 - 1 = \text{FF}_{16} = 11111111_2$
• Back in the old days: some databases could only handle strings up to 255 bytes
• MySQL (without innodb_large_prefix) limits the index key to 767 bytes: 255 characters * 3 bytes for UTF-8 = 765 bytes
Agenda

• Text Types
• **Numeric Types**
• Dates and Times
• XML
• JSON
• Boolean
• Bits
• Binary Data
• Network
• Create your own Data Type
Numeric Types

- Integer (Smallint / INT2, Integer / INT4, Bigint / INT8)
- Floating Point (Real, Double Precision)
- Numeric
- Sequence (Smallserial, Serial, Bigserial)
## Numeric Types: Integers

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT / INT2</td>
<td>2 Bytes</td>
<td>-32.768 to +32.767</td>
</tr>
<tr>
<td>INTEGER / INT4</td>
<td>4 Bytes</td>
<td>-2.147.483.648 to +2.147.483.647</td>
</tr>
<tr>
<td>BIGINT / INT8</td>
<td>8 Bytes</td>
<td>-9.223.372.036.854.775.808 to +9.223.372.036.854.775.807</td>
</tr>
</tbody>
</table>

Note: Alignment might ruin your day:

Smallint / Integer / Smallint / Integer = 16 Bytes
Smallint / Smallint / Integer / Integer = 12 Bytes
## Numeric Types: Floating Point

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td>4 Bytes</td>
<td>6 decimal digits</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>8 Bytes</td>
<td>15 decimal digits</td>
</tr>
</tbody>
</table>

Note: Values can be inaccurate (rounded), even if shown exact
Numeric Types: Floating Point

SELECT '100001'::REAL AS real;
real
--------
100001

SELECT '10000001'::REAL AS real;
real
--------
1e+07

SELECT '100001.5'::REAL AS real;
real
--------
100002

6 decimal digits
7 decimal digits
6+1 decimal digits
Numeric Types: Floating Point

- `SELECT '100000000000001'::DOUBLE PRECISION AS double;`
  - double: 100000000000001
  - 15 decimal digits

- `SELECT '1000000000000001'::DOUBLE PRECISION AS double;`
  - double: 1e+15
  - 16 decimal digits

- `SELECT '100000000000001.5'::DOUBLE PRECISION AS double;`
  - double: 1000000000000002
  - 15+1 decimal digits
Numeric Types: Floating Point

Conclusions:

• Floating point numbers are imprecise
• Never to use for exact values (like €€€ or $$$)
• Ok for something like gauges in monitoring (but better round the result)
Numeric Types: Numeric

- Up to 1000 numbers precision
- Definition: NUMERIC(10, 3) = 1234567.123
- Handled in software (no hardware support)
- Storage requirements: 2 bytes per 4 decimal digits, plus 3-8 bytes overhead
- NUMERIC(? , 20) is the default
Do you know Sissa ibn Dahir?

Hint: lived in India, in 3rd or 4th century
The king’s name at this time was: Shihram
**Numeric Types: Numeric**

Number of rice grains:

\[1+2+2^2+2^4\ldots+2^{63} = 2^{64} - 1\]

```sql
SELECT power(2::DOUBLE PRECISION, 64::DOUBLE PRECISION) - 1;
?column?
---------------------
1.84467440737096e+19
```

```sql
SELECT power(2::NUMERIC, 64::NUMERIC) - 1;
?column?
--------------------------------------
18446744073709551615.0000000000000000
```

20 decimal digits (980 left)
Money Type

PostgreSQL has a Money type:

- Only one currency ($lc_monetary), always shown
- Can be represented with NUMERIC + formatting as well
- Uses 8 bytes of storage
- It will round fractions!
- Handles: -92233720368547758.08 to +92233720368547758.07 (92 Quadrillion)
- Current Brazil debt (today): R$ 6,600,361,420,917 (R$ 6,6 Trillion)
- Maybe 2 users in the world
- Deprecated twice, resurrected
### Data Types: Sequences

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Numeric Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLSERIAL</td>
<td>2 Bytes</td>
<td>INT2</td>
</tr>
<tr>
<td>SERIAL</td>
<td>4 Bytes</td>
<td>INT4</td>
</tr>
<tr>
<td>BIGSERIAL</td>
<td>8 Bytes</td>
<td>INT8</td>
</tr>
</tbody>
</table>

- Sequences start (by default) with “1”
- Step “1” (by default) forward (by default)
- Sequence can cycle (default: no)
- Sequence name can be used in multiple tables
- Sequence can only be owned by one table
- Sequence is NOT transactional
Data Types: Sequences

SELECT currval('my_sequence'); -- current value
    currval
--------
      23

SELECT nextval('my_sequence'); -- next value
    nextval
--------
      24

Sequence must have been used before in the current session.
Data Types: Sequences

SELECT setval('my_sequence', 50); -- set new value

SELECT setval('my_sequence',
(SELECT MAX(id)
    FROM table));
### Data Types: Sequences

**CREATE TABLE** `public.seq` (  
  `id` `SERIAL` `PRIMARY KEY`  
);  

```sql  
SELECT `pg_get_serial_sequence`('public.seq', 'id');  
```

```
pg_get_serial_sequence
------------------------
public.seq_id_seq
(1 row)
```
# Data Types: Sequences

```sql
SELECT * FROM public.seq_id_seq;
```

<table>
<thead>
<tr>
<th>sequence_name</th>
<th>seq_id_seq</th>
<th>last_value</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_value</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>increment_by</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max_value</td>
<td>9223372036854775807</td>
<td></td>
<td></td>
</tr>
<tr>
<td>min_value</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cache_value</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>log_cnt</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_cycled</td>
<td>f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_called</td>
<td>f</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Agenda

• Text Types
• Numeric Types
• **Dates and Times (Temporal Types)**
• XML
• JSON
• Boolean
• Bits
• Binary Data
• Network
• Create your own Data Type
Question

• What’s the time at South Pole right now?
Date and Time Types (Temporal Types)

- **TIMESTAMP WITHOUT TIME ZONE** (short: TIMESTAMP): stores date and time
- **TIMESTAMP WITH TIME ZONE** (short: TIMESTAMPTZ): stores date and time plus time zone
- **TIME WITHOUT TIME ZONE** (short: TIME): stores a time
- **TIME WITH TIME ZONE** (short: TIMETZ): stores a time plus time zone
- **DATE**: stores a date
- **INTERVAL**: stores a time difference (between two times)

- Note: TZ types will deal with DST
- Note: will NOT deal with leap seconds
Date Ranges

• Timestamp: 4713 BC – 294276 AD
• Date: 4713 BC – 5874897 AD
• No year “0” (1 BC is followed by 1 AD)
Timestamp Without Time Zone

• Don’t use that to store UTC times (that is, times with time zone)
• Only use that if the actual date/time has no time zone
Time With Time Zone

• How useful is a time with a time zone?
• Use TIMESTAMPTZ instead
• Only there because required by the SQL standard
Date and Time examples

SELECT '2018-10-11'::TIMESTAMP; -- simple timestamp

timestamp
-------------------
2018-10-11 00:00:00

SELECT 'January 5 2019'::TIMESTAMP; -- silly US format

timestamp
-------------------
2019-01-05 00:00:00

Shown as time, because of the TIMESTAMP cast
Date and Time examples

SELECT '2018-08-10 03:25:00 PM UTC'::TIMESTAMPTZ; -- Summer

2018-08-10 17:25:00+02

SELECT '2018-12-12 10:23:00 UTC'::TIMESTAMPTZ; -- Winter

2018-12-12 11:23:00+01

Shown as local time (to the database server)
Date and Time examples

SELECT ‘2018-04-12 20:00:00 America/Sao_Paulo’::TIMESTAMPZ;
timestamp
-----------------------
2018-04-13 01:00:00+02

SELECT ‘2018-04-12 20:00:00 -3’::TIMESTAMPZ;
timestamp
-----------------------
2018-04-13 01:00:00+02

Time Zone number
No DST handling

Time zone name
DTS handling
Date and Time examples

BEGIN;
SELECT NOW();
now
---------------------------------

SET TIME ZONE 'America/Sao_Paulo';

SELECT NOW();
now
---------------------------------
2019-01-11 08:55:57.162307+03
Date and Time examples

SELECT NOW() AT TIME ZONE 'America/Sao_Paulo';
now
-------------------------------------

Just for this query
Interval examples

```sql
SELECT '2000-01-05'::TIMESTAMP - '2000-01-01'::TIMESTAMP;
?column?
--------
4 days
(1 row)
```

```sql
SELECT '2000-01-01'::TIMESTAMP - '2000-01-04'::TIMESTAMP;
?column?
--------
-3 days
```
Interval examples

```sql
SELECT '2018-10-23 00:23:12'::TIMESTAMP - '2018-10-12 07:05:25'::TIMESTAMP;
```

?column?

--------------

10 days 17:17:47
Interval examples

```
SELECT '2000-02-28 00:00:00'::TIMESTAMP + INTERVAL '1 day 02:00:00';
?column?
```

-------------------

2000-02-29 02:00:00

Leap year
Interval examples

```
SELECT '2001-01-01'::DATE - '2000-01-01'::DATE;
?column?
---------
366
```

2000 is a leap year

```
SELECT '2002-01-01'::DATE - '2001-01-01'::DATE;
?column?
---------
365
```

2001 is not a leap year
What’s the time at South Pole?

- In theory, North Pole and South Pole have all times of the day
- Depending on the direction where one is looking
- Amundsen-Scott Station (USA) is supplied from New Zealand
- Therefore they use the same time zone (NZ – New Zealand)
Date and Time types: time at South Pole

```sql
SELECT NOW() AT TIME ZONE 'NZ';
now
----------------------------------
2019-01-12 01:55:57.162307
```
Date and Time types: time at South Pole

```sql
SELECT NOW() AT TIME ZONE 'Antarctica/South_Pole';
```

now
---------------------------
2019-01-12 01:55:57.162307

Depends on what time zones your OS knows
Agenda

• Text Types
• Numeric Types
• Dates and Times
• XML
• JSON
• Boolean
• Bits
• Binary Data
• Network
• Create your own Data Type
XML Type

- XML (Extensible Markup Language) defines a document structure
- Hot stuff from the 90s … so last century
- PostgreSQL does simple validation (like correct syntax), but no DTD validation
- Content can be a XML document, or a XML fragment
- Encoding is assumed to be in “client_encoding”, encoding specification in XML is ignored
- Exception: binary mode (encoding specification is observed, or UTF-8 is assumed)
- It is not possible to directly search in XML types
XML Type

SELECT XMLPARSE (DOCUMENT '<?xml version="1.0"?
><database><name>PostgreSQL</name><vendor>PostgreSQL Global Development Group</vendor></database>');

xmlparse

<database><name>PostgreSQL</name><vendor>PostgreSQL Global Development Group</vendor></database>
XML Type

```sql
SELECT XMLPARSE (CONTENT '<name>PostgreSQL</name>');
```

```
--

<name>PostgreSQL</name>
```
**XML Type**

```sql
SELECT XMLSERIALIZE (CONTENT '<name>PostgreSQL</name>' AS TEXT);

xmlserialize
-------------------------

<name>PostgreSQL</name>
```

That's a string now
Agenda

- Text Types
- Numeric Types
- Dates and Times
- XML
- JSON
- Boolean
- Bits
- Binary Data
- Network
- Create your own Data Type
JSON Type

- JSON (JavaScript Object Notation) defines an open format to exchange attribute-value pairs
- Used in many web frameworks and IoT data exchange, among others
- Many NoSQL databases use JSON as native format
- PostgreSQL offers two JSON data types:
  - JSONB: stores data in decomposed binary, supports indexing
  - JSON: stores raw data, must be processed on each request
- Uses regular transactions
JSONB Type

SELECT '"abc"'::jsonb;

jsonb
-------
"abc"

Extra quotes for text in JSON
JSONB Type

SELECT '
[ "abc", "def", "ghi"]'::jsonb;

jsonb

-----------------------
[ "abc", "def", "ghi"]

Array

SELECT '{"1": "abc", "2": "def", "3": "ghi"}'::jsonb;

jsonb

--------------------------------------
{ "1": "abc", "2": "def", "3": "ghi"}

Key/Value Pairs
JSONB Type

SELECT '{"1": "abc", "2": "def", "3": "ghi"} '::jsonb->'2'::
  ?column?
-----------------
  "def"

Access Element with key “2”
SELECT '"abc", "def", "ghi"'::jsonb @> '"ghi"'::jsonb;
?column?

---------

 t

Is the right element in the left?
JSONB Type

SELECT '['"abc", "def", "ghi"]'::jsonb ? 'def';
?column?

SELECT '{"1": "abc", "2": "def", "3": "ghi"}'::jsonb ? '2';
?column?
JSON Type with GIN index

- The GIN index supports JSON queries
- Only works with JSONB, not the JSON type
JSON Type with GIN index

CREATE INDEX idx_gin ON nosqltable USING gin ((data->'name'));

SELECT * FROM nosqltable
WHERE data->'name' ? 'Scherbaum';
Agenda

• Text Types
• Numeric Types
• Dates and Times
• XML
• JSON
• Boolean
• Bits
• Binary Data
• Network
• Create your own Data Type
Boolean Type

• PostgreSQL supports a real Boolean type – please use it!

• Values for **True**: TRUE, true, 1, ’t’, ’y’ and ’yes’
• Values for **False**: FALSE, false, 0, ’f’, ’n’ and ’no’
Boolean Type

SELECT true::BOOLEAN;
bool
-------
t

SELECT false::BOOLEAN;
bool
-------
f
CREATE TABLE logging_table ( 
    id INTEGER NOT NULL PRIMARY KEY, 
    content FLOAT NOT NULL, 
    error BOOLEAN NOT NULL 
);

INSERT INTO logging_table (id, content, error) 
SELECT generate_series (1, 1000000), RANDOM(), 
    CASE WHEN RANDOM() < 0.02 
        THEN TRUE ELSE FALSE END;
Boolean Type: Partial Index

```
EXPLAIN SELECT COUNT(*) FROM logging_table WHERE error = TRUE;

QUERY PLAN

--------------------------------------------------------------
Aggregate  (cost=16422.42..16422.43 rows=1 width=0)
    ->  Seq Scan on logging_table (cost=0.00..16370.00
                                  rows=20967 width=0)
          Filter: error
(3 rows)
```
Boolean Type: Partial Index

CREATE INDEX partial_index
    ON logging_table (error)
    WHERE error = TRUE;

EXPLAIN SELECT COUNT(*) FROM logging_table WHERE error = TRUE;

QUERY PLAN

--------------------------------------------------------------
Aggregate  (cost=68.41..68.42 rows=1 width=0)
    ->  Index Only Scan using partial_index on
        logging_table (cost=0.29..15.99 rows=20967 width=0)
            Index Cond: (error = true)
(3 rows)
Boolean Type: Partial Index

CREATE INDEX full_index
    ON logging_table (error);

SELECT pg_relation_size('logging_table') / 8192 AS "Pages",
    pg_relation_size('partial_index') / 8192
    AS "Pages partial Index",
    pg_relation_size('full_index') / 8192
    AS "Pages full Index";

<table>
<thead>
<tr>
<th>Pages</th>
<th>Pages partial Index</th>
<th>Pages full Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>6370</td>
<td>57</td>
<td>2745</td>
</tr>
</tbody>
</table>
Agenda

- Text Types
- Numeric Types
- Dates and Times
- XML
- JSON
- Boolean
- Bits
- Binary Data
- Network
- Create your own Data Type
Bits

- BIT(n): stores a bit string with length ‘n’
- BIT VARYING(n): stores a bit string up to the length of ‘n’
- BIT: equals BIT(1)
- Logical operations like AND, OR, XOR possible
CREATE TABLE bit_test (id SERIAL, data BIT(5));

INSERT INTO bit_test (data) VALUES (B’10101’);

SELECT id, data FROM bit_test;

<table>
<thead>
<tr>
<th>id</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10101</td>
</tr>
</tbody>
</table>
Bits

```
SELECT id, data & B'00001' FROM bit_test;
  id | data
-----+------
  1  | 00001

SELECT id, data | B'01011' FROM bit_test;
  id | data
-----+------
  1  | 11111
```
Bits

SELECT id, data # B'11111' FROM bit_test;

| id | data |
|----+------|
| 1  | 01010 |

SELECT id, data << 1, data FROM bit_test;

| id | ?column? | data |
|----+----------+------|
| 1  | 101010    | 10101 |

42

21

XOR

Shift left: \( \times 2 \)
Bits

```
SELECT id, data FROM bit_test
WHERE (data & B'00001')::INTEGER > 0;
  id | data
----+------
    1  | 10101

SELECT id, data FROM bit_test
WHERE (data & B'00010')::INTEGER > 0;
  id | data
----+------
    1  | 10101
```

Search for Bit
### Bits

```sql
SELECT 23::BIT(5);
  bit
-------
10111
```

Cast INT to BIT

```sql
SELECT B'10101'::BIT(5)::INTEGER, X'FE'::BIT(8)::INTEGER;
  int4 | int4
------|------
  21   | 254
```
Agenda

- Text Types
- Numeric Types
- Dates and Times
- XML
- JSON
- Boolean
- Bits
- Binary Data
- Network
- Create your own Data Type
Binary Data: ByteA

- Binary data (unprintable characters) can’t be stored in TEXT types
- Data which does not fit into the available encodings can’t be stored in TEXT types
- Binary data might contain 0 bytes (no bits set), however that is the “end of string” sign in C
- PostgreSQL offers **ByteA** for binary data
- PostgreSQL understands 2 output formats: HEX (new) and ESCAPE (old)
- Please use functions in your programming language to transfer data
SET bytea_output TO hex;
SELECT E'\000':bytea;

SET bytea_output TO escape;
SELECT E'\000':bytea;

\x00

\000

The E is for Escape
Agenda

• Text Types
• Numeric Types
• Dates and Times
• XML
• JSON
• Boolean
• Bits
• Binary Data
• Network
• Create your own Data Type
## Data Types: Network Address Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Storage Size</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>INET</td>
<td>7 / 19 Bytes</td>
<td>IPv4 / IPv6 host/network</td>
</tr>
<tr>
<td>CIDR</td>
<td>7 / 19 Bytes</td>
<td>IPv4 / IPv6 network</td>
</tr>
<tr>
<td>MACADDR</td>
<td>6 Bytes</td>
<td>MAC Ethernet address</td>
</tr>
</tbody>
</table>

- Uses classless routing convention
Data Types: Network Address Types

SELECT '192.168.0.1/24'::INET; -- store address and network
  inet
----------------
192.168.0.1/24

SELECT '192.168.0.1'::CIDR; -- assume network mask
  cidr
----------------
192.168.0.1/32

SELECT '192.168.5'::CIDR; -- assume network mask
  cidr
----------------
192.168.5.0/24
Data Types: Network Address Types

CREATE TABLE idr ( 
    idr    INET    PRIMARY KEY 
);

CREATE INDEX idr_idr ON idr(idr);

INSERT INTO idr (idr) 
    VALUES ('192.168.0.1'), ('192.168.0.99'), ('10.0.0.1');
Data Types: Network Address Types

SELECT * FROM idr
    WHERE idr <<= '192.168.0.0/24'::CIDR;

idr
---
192.168.0.1
192.168.0.99

Limit to certain network range
Data Types: Network Address Types

```
EXPLAIN SELECT * FROM idr
    WHERE idr << '192.168.0.0/24'::CIDR;
```

**QUERY PLAN**

```
Bitmap Heap Scan on idr (cost=4.22..14.37 rows=1 width=32)
    Filter: (idr << '192.168.0.0/24'::inet)
    -> Bitmap Index Scan on idr_idr (cost=0.00..4.22 rows=7 width=0)
       Index Cond: ((idr > '192.168.0.0/24'::inet) AND
                      (idr <= '192.168.0.255'::inet))
```

Will use Index
Agenda

• Text Types
• Numeric Types
• Dates and Times
• XML
• JSON
• Boolean
• Bits
• Binary Data
• Network
• Create your own Data Type
Create your own data type

- Composite type
- Enumerations (ENUM)
- WYODT – Write your own base data type
- Use EXTENSIONs (like PostGIS)
Enumerations

- Predefined list with values
- List is (should) not to change
- If the list is to change, consider a 1:n table (lookup table) instead
Question

- ENUM is often used for gender
- How many different gender value do you know?
Gender types

- male/female
- unknown
- hybrid
- today male/female
- denied (different from “unknown”)
- not applicable

- See ISO/IEC 5218
- Facebook currently allows 56 different gender types
- Conclusion: Think beforehand if your data type really is an ENUM or might change in the future.
Enumerations

CREATE TYPE card_colors
    AS ENUM ('Diamonds', 'Hearts', 'Spades', 'Clubs');

CREATE TABLE card_results ( id SERIAL PRIMARY KEY,
    color card_colors NOT NULL,
    winner TEXT NOT NULL
);
Enumerations

```
INSERT INTO card_results (color, winner)
    VALUES ('Hearts', 'Paul');
INSERT INTO card_results (color, winner)
    VALUES ('Diamonds', 'Jim');

SELECT id, color, winner
    FROM card_results
    WHERE color = 'Hearts';
```

```
<table>
<thead>
<tr>
<th>id</th>
<th>color</th>
<th>winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hearts</td>
<td>Paul</td>
</tr>
</tbody>
</table>
```
What’s missing?

- Arrays
- Composite Types
- Range Types
- Geometric Types
- UUID Type
- OID
- Create your very own types (write some code)