



Tour de (PostgreSQL) Data Types

Andreas Scherbaum

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

Target audience for this talk

This talk is for you

- Migrate from another database
- Basic experience with data types
- Want to learn something new
- Just want a seat for ~~Simon's~~ the next talk

This talk is not for you

- Read -hackers daily
- Use more than 7-10 different data types

Simon's talk about "Replication & Recovery" is cancelled

Replacement: Semantic Web with PostgreSQL

Data Types in PostgreSQL

Quick poll: 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
```

> 0 references another type

```
AND typrelid = 0;
```

> 0 references pg_class
(table types)

Number Data Types

82

Data Types in PostgreSQL

```
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 timestamptz 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

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(n)]		fixed-length bit string
bit varying [(n)]	varbit	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data ("byte array")
character [(n)]	char [(n)]	fixed-length character string
character varying [(n)]	varchar [(n)]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [fields] [(p)]		time span
json		textual JSON data
jsonb		binary JSON data, decomposed
line		infinite line on a plane
lseg		line segment on a plane
macaddr		MAC (Media Access Control) address
money		currency amount
numeric [(p, s)]	decimal [(p, s)]	exact numeric of selectable precision
path		geometric path on a plane
pg_lsn		PostgreSQL Log Sequence Number
point		geometric point on a plane
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(p)] [without time zone]		time of day (no time zone)
time [(p)] with time zone	timetz	time of day, including time zone
timestamp [(p)] [without time zone]		date and time (no time zone)
timestamp [(p)] with time zone	timestamptz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot
uuid		universally unique identifier
xml		XML data

General-purpose data types: 41

Data Types in PostgreSQL

Another poll: how many different data types are you using?

Agenda

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

Agenda

- **Text Types**
- Numeric Types
- Dates and Times
- XML
- JSON
- Boolean
- Bits
- Binary Data
- Network
- Arrays
- 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 whitespaces
- CHAR: 1 byte string
- CHAR(n): string with length 'n'
- TEXT: string up to ~1GB

Text Types: VARCHAR versus CHAR

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

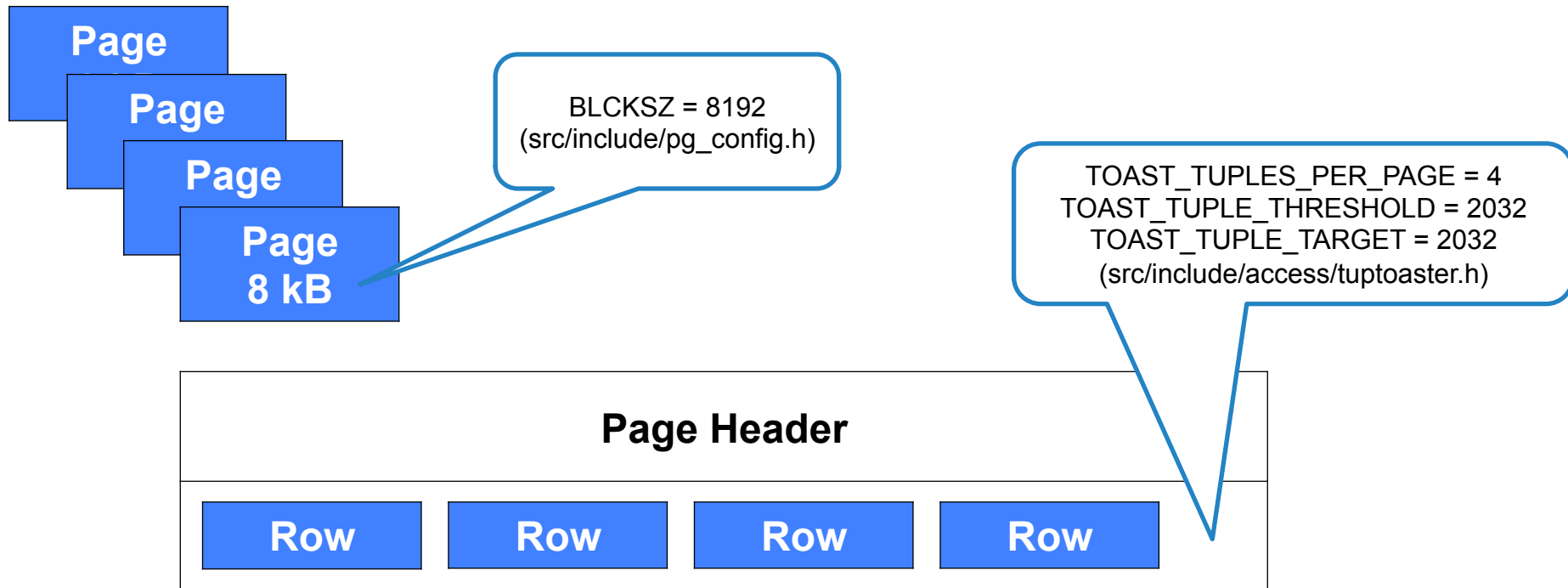
5x Whitespace

vc_1	vc_5	vc_10	c_1	c_5	c_10
1	5	5	1	5	10

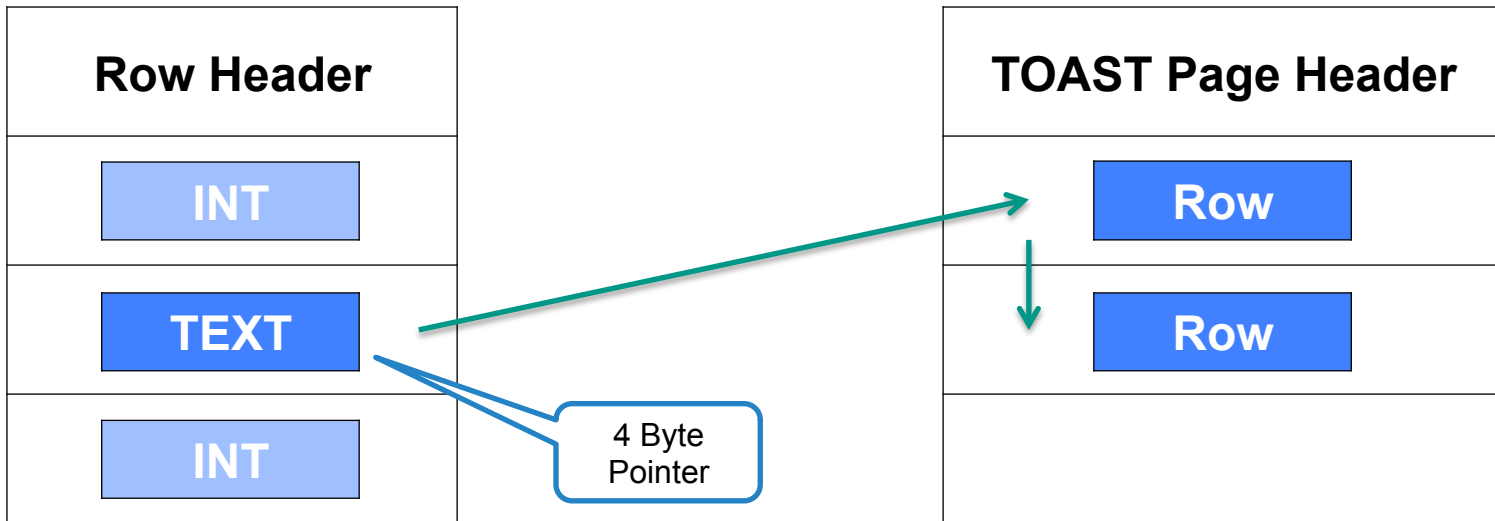
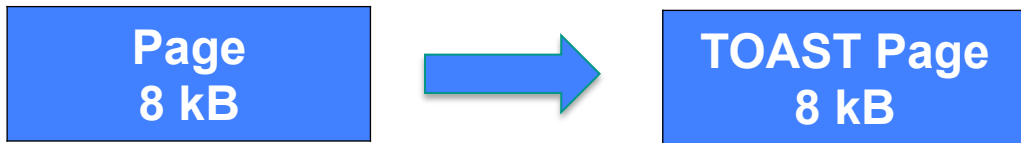
(1 row)

LENGTH() and CHAR_LENGTH() return '0'

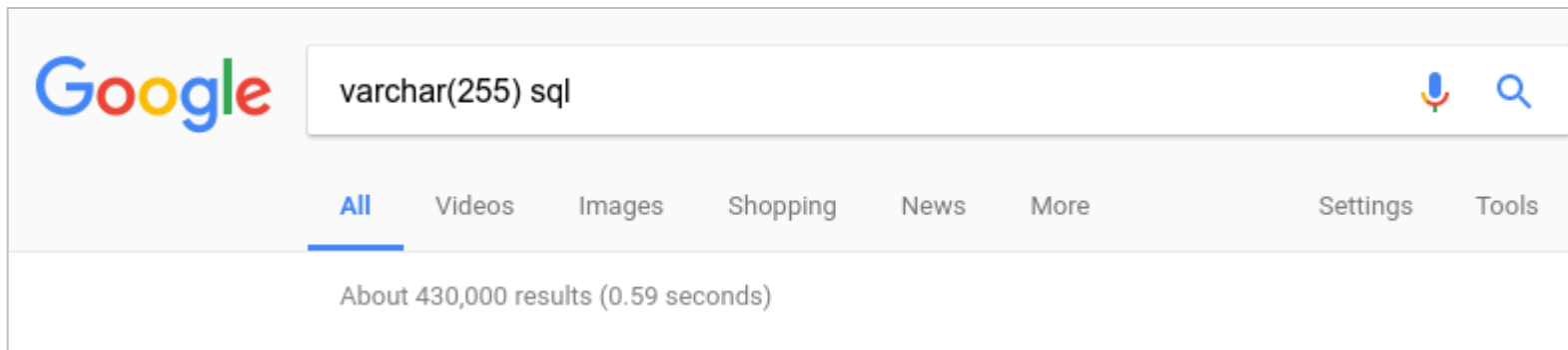
Internals: Pages & TOAST



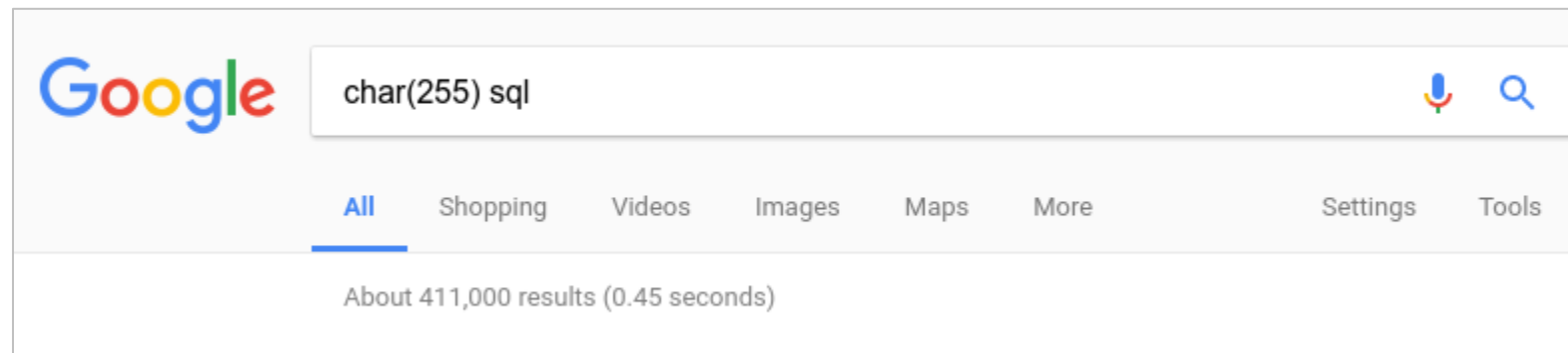
Internals: Pages & TOAST



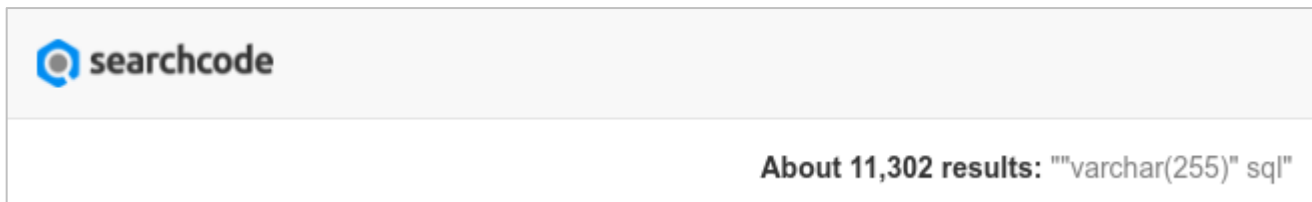
What about CHAR(255)?



Google search results for "varchar(255) sql". The search bar contains the text "varchar(255) sql". Below the search bar, the "All" tab is selected. The results show "About 430,000 results (0.59 seconds)".



Google search results for "char(255) sql". The search bar contains the text "char(255) sql". Below the search bar, the "All" tab is selected. The results show "About 411,000 results (0.45 seconds)".



searchcode search results for "varchar(255) sql". The search bar contains the text "varchar(255) sql". Below the search bar, the results show "About 11,302 results: ""varchar(255)" sql"".

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
- Arrays
- 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

Name	Storage Size	Range
SMALLINT / INT2	2 Bytes	-32.768 to +32.767
INTEGER / INT4	4 Bytes	-2.147.483.648 to +2.147.483.647
BIGINT / INT8	8 Bytes	-9.223.372.036.854.775.808 to +9.223.372.036.854.775.807

Note: Alignment might ruin your day:

Smallint / Integer / Smallint / Integer = 16 Bytes

Smallint / Smallint / Integer / Integer = 12 Bytes

Numeric Types: Floating Point

Name	Storage Size	Precision
REAL	4 Bytes	6 decimal digits
DOUBLE PRECISION	8 Bytes	15 decimal digits

Note: Values can be inaccurate (rounded), even if shown exact

Numeric Types: Floating Point

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

```
real
```

```
-----
```

```
100001
```

6 decimal digits

```
SELECT '10000001'::REAL AS real;
```

```
real
```

```
-----
```

```
1e+07
```

7 decimal digits

```
SELECT '100001.5'::REAL AS real;
```

```
real
```

```
-----
```

```
100002
```

6+1 decimal digits

Numeric Types: Floating Point

```
SELECT '1000000000000001'::DOUBLE PRECISION AS double;
```

double

15 decimal digits

1000000000000001

```
SELECT '1000000000000001'::DOUBLE PRECISION AS double;
```

double

16 decimal digits

1e+15

```
SELECT '1000000000000001.5'::DOUBLE PRECISION AS double;
```

double

15+1 decimal digits

1000000000000002

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)

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
- Handles: -92233720368547758.08 to +92233720368547758.07 (92 Quadrillion)
- Current US depth (Jan 2017): 19,939,760,263,983.42 (\$19 Trillion)
- Maybe 2 users in the world
- Deprecated twice, resurrected

Numeric Types: Numeric

- Up to 1000 numbers precision
- Definition: `NUMERIC(10, 3) = 1234567.123`
- Handled in software (no hardware support)

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:

$$1+2+2^2+2^4\dots2^{63} = 2^{64} - 1$$

```
SELECT power(2::DOUBLE PRECISION, 64::DOUBLE PRECISION) - 1;  
?column?
```

1.84467440737096e+19

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

18446744073709551615.0000000000000000



20 decimal digits (980 left)

Data Types: Sequences

Name	Storage Size	Numeric Type
SMALLSERIAL	2 Bytes	INT2
SERIAL	4 Bytes	INT4
BIGSERIAL	8 Bytes	INT8

- 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

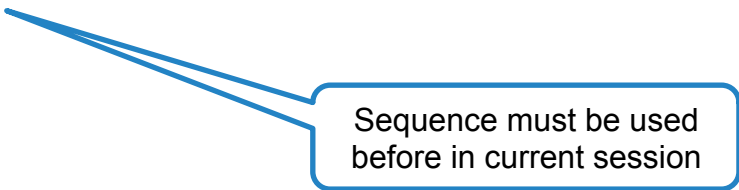
Data Types: Sequences

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

```
currval
```

```
-----
```

```
23
```



Sequence must be used
before in current session

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

```
nextval
```

```
-----
```

```
24
```

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  
);
```

```
SELECT pg_get_serial_sequence('public.seq', 'id');  
       pg_get_serial_sequence  
-----  
       public.seq_id_seq  
(1 row)
```

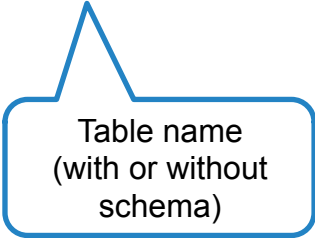


Table name
(with or without
schema)



Column name

Data Types: Sequences

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

```
-[ RECORD 1 ]-+-----
```

sequence_name		seq_id_seq
last_value		1
start_value		1
increment_by		1
max_value		9223372036854775807
min_value		1
cache_value		1
log_cnt		0
is_cycled		f
is_called		f

Agenda

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

What's shown in this picture?



Question

- What's the time at South Pole right now?

Date and Time 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 and Time examples

```
SELECT '2016-10-11'::TIMESTAMP; -- simple timestamp  
timestamp
```

2016-10-11 00:00:00

```
SELECT 'January 5 2017'::TIMESTAMP; -- silly US format  
timestamp
```

2017-01-05 00:00:00



Shown as time, because
of the TIMESTAMP cast

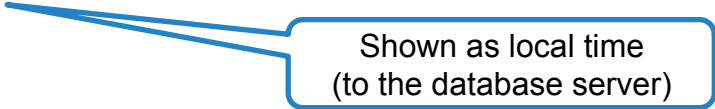
Date and Time examples

```
SELECT '2016-08-10 03:25:00PM UTC'::TIMESTAMPTZ; -- Summer  
timestamptz
```

```
2016-08-10 17:25:00+02
```

```
SELECT '2016-12-12 10:23:00 UTC'::TIMESTAMPTZ; -- Winter  
timestamptz
```

```
2016-12-12 11:23:00+01
```



Shown as local time
(to the database server)

Date and Time examples

```
SELECT '2016-04-12 00:00:00 Europe/Moscow'::TIMESTAMPTZ;  
timestamptz
```

```
2016-04-11 22:00:00+02
```

Time Zone number
No DST handling

Time zone name
DTS handling

```
SELECT '2016-04-12 00:00:00 +4'::TIMESTAMPTZ;  
timestamptz
```

```
2016-04-11 22:00:00+02
```

Date and Time examples

```
BEGIN;
```

```
SELECT NOW();
```

```
now
```

Transaction stops time

2017-01-11 13:55:57.162307+01

```
SET TIME ZONE 'Europe/Moscow';
```

```
SELECT NOW();
```

```
now
```


2017-01-11 15:55:57.162307+03

Date and Time examples

```
SELECT NOW() AT TIME ZONE 'Europe/Moscow';
```

now

```
2017-01-11 15:55:57.162307
```



Just for this query

Interval examples

```
SELECT '2000-01-05'::TIMESTAMP - '2000-01-01'::TIMESTAMP;
```

```
?column?
```

```
-----
```

```
4 days
```

```
(1 row)
```



Interval

```
SELECT '2000-01-01'::TIMESTAMP - '2000-01-04'::TIMESTAMP;
```

```
?column?
```

```
-----
```

```
-3 days
```


Interval examples

```
SELECT '2016-10-23 00:23:12'::TIMESTAMP -  
'2016-10-12 07:05:25'::TIMESTAMP;
```

?column?

10 days 17:17:47



Interval

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

```
SELECT NOW() AT TIME ZONE 'NZ';
```

now

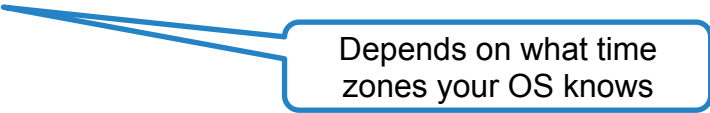
2017-01-12 01:55:57.162307

Date and Time types: time at South Pole

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

now

2017-01-12 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
- Arrays
- Create your own Data Type

XML Type

- XML (Extensible Markup Language) defines a document structure
- Hot stuff from the 90s ... 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

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

<name>PostgreSQL</name>

XML Type

```
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
- Arrays
- 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

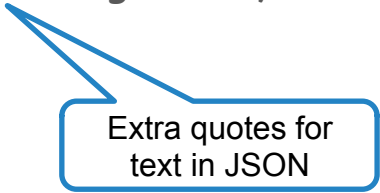
JSONB Type

```
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"}

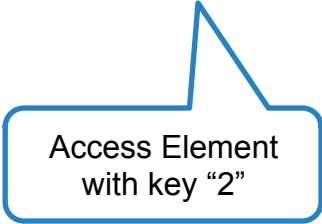
Key/Value Pairs

JSONB Type

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

```
?column?
```

```
-----  
"def"
```

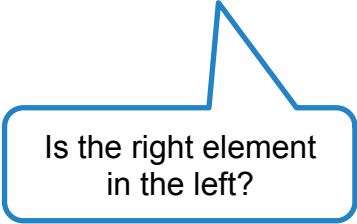


Access Element
with key "2"

JSONB Type

```
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?
```

```
-----
```

```
t
```

Is the right value
in the left data?

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

```
?column?
```

```
-----
```

```
t
```

Is the right key
in the left data?

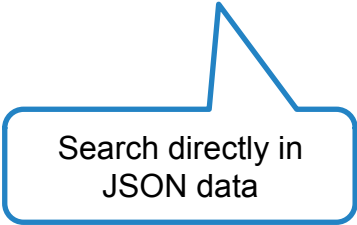
JSON Type with GIN index

- 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';
```



Search directly in
JSON data

Agenda

- Text Types
- Numeric Types
- Dates and Times
- XML
- JSON
- **Boolean**
- Bits
- Binary Data
- Network
- Arrays
- 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
```

Boolean Type: Partial Index

```
CREATE TABLE boolean_index (  
  id            INTEGER          NOT NULL PRIMARY KEY,  
  content       FLOAT           NOT NULL,  
  error        BOOLEAN        NOT NULL  
);
```

```
INSERT INTO boolean_index (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 boolean_index WHERE error = TRUE;  
QUERY PLAN
```

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

Boolean Type: Partial Index

```
CREATE INDEX planer_index_test_fehler  
    ON boolean_index (error)  
WHERE error = TRUE;
```

```
EXPLAIN SELECT COUNT(*) FROM boolean_index WHERE error = TRUE;  
          QUERY PLAN
```

```
-----  
Aggregate  (cost=68.41..68.42 rows=1 width=0)  
->  Index Only Scan using planer_index_test_fehler on  
    boolean_index  (cost=0.29..15.99 rows=20967 width=0)  
        Index Cond: (error = true)  
  
(3 rows)
```

Boolean Type: Partial Index

```
CREATE INDEX planer_index_test_fehler_komplett  
ON boolean_index (error);
```

```
SELECT pg_relation_size('boolean_index') / 8192 AS "Pages",  
       pg_relation_size('planer_index_test_fehler') / 8192 AS  
         "Pages partial Index",  
       pg_relation_size('planer_index_test_fehler_komplett') /  
         8192 AS "Pages full Index";
```

Pages	 	Pages partial Index	 	Pages full Index
6370		57		2745

Agenda

- Text Types
- Numeric Types
- Dates and Times
- XML
- JSON
- Boolean
- **Bits**
- Binary Data
- Network
- Arrays
- 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

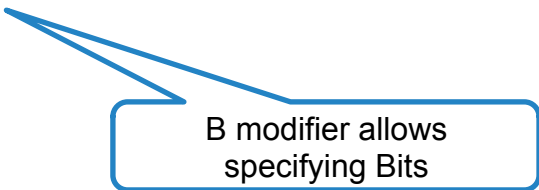
Bits

```
CREATE TABLE bit_test (id SERIAL, data BIT(5));
```

```
INSERT INTO bit_test (data) VALUES (B'10101');
```

```
SELECT id, data FROM bit_test;
```

id		data
-----+-----		
1		10101



B modifier allows
specifying Bits

Bits

```
SELECT id, data & B'00001' FROM bit_test;
```

id	data
1	00001

AND

```
SELECT id, data | B'01011' FROM bit_test;
```

id	data
1	11111

OR

Bits

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

id	data
1	01010

XOR

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

id	?column?	data
1	101010	10101

Shift left: * 2

42

21

Bits

```
SELECT id, data FROM bit_test
WHERE (data & B'00001')::INTEGER > 0;
```

id	data
1	10101



Search for Bit

```
SELECT id, data FROM bit_test
WHERE (data & B'00010')::INTEGER > 0;
```

id	data
----	------

Bits

```
SELECT 23::BIT(5);
```

```
bit
```

```
-----
```

```
10111
```



Cast INT to BIT

```
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
- Arrays
- Create your own Data Type

Binary Data: ByteA

- Binary data (unprintable characters) 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

Binary Data: ByteA

```
SET bytea_output TO hex;
```

```
SELECT E'\\000'::bytea;
```

```
bytea
```

```
-----
```

```
 \x00
```

```
SET bytea_output TO escape;
```

```
SELECT E'\\000'::bytea;
```

```
bytea
```

```
-----
```

```
 \000
```

Agenda

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

Data Types: Network Address Types

Name	Storage Size	Stores
INET	7 / 19 Bytes	IPv4 / IPv6 host/network
CIDR	7 / 19 Bytes	IPv4 / IPv6 network
MACADDR	6 Bytes	MAC Ethernet address

- 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');
```

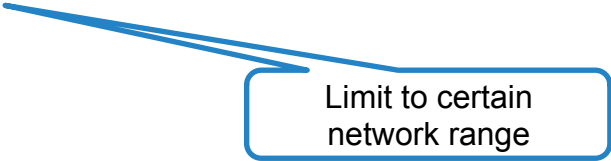
Data Types: Network Address Types

```
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
- **Arrays**
- Create your own Data Type

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)

Arrays

```
SELECT ARRAY[['abc', 'def'], ['123', '[456]']];
```

array

{ {abc, def}, {123, [456]} }



2 x 2 dimensions

```
SELECT array_dims(ARRAY[['abc', 'def'], ['123', '[456]']]);
```

array_dims

[1:2][1:2]

Arrays

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

array

abc



1st Element

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

array

{def,ghi}



Element 2 to 3

Arrays

```
SELECT array_dims(ARRAY[['abc', 'def'], ['123', '[456']]);
```

```
array_dims
```

```
-----  
[1:2][1:2]
```

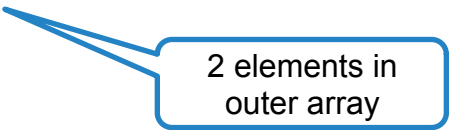


2 x 2 Dimensions

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

```
array_length
```

```
-----  
2
```



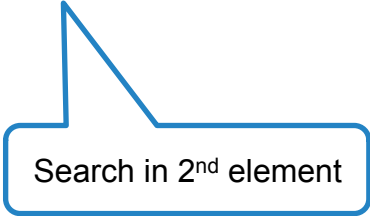
2 elements in
outer array

Arrays

```
SELECT TRUE WHERE (ARRAY[ 'abc', 'def', 'ghi' ])[2] = 'def';
```

bool

t



Search in 2nd element

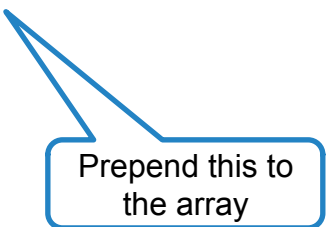
Arrays

```
SELECT array_prepend(0, ARRAY[1, 2, 3]);
```

```
array_prepend
```

```
-----  
{0,1,2,3}
```

```
(1 row)
```

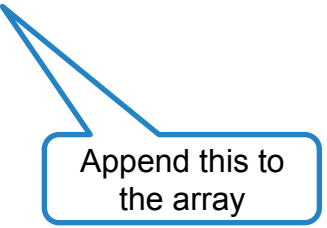


Prepend this to
the array

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

```
array_append
```

```
-----  
{1,2,3,4}
```



Append this to
the array

Arrays

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

```
array_to_string
```

```
-----
```

```
abc - def - ghi
```

```
(1 row)
```

Fill in between
array elements

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

```
unnest
```

```
-----
```

```
abc
```

```
def
```

```
ghi
```

Transform array
into rows

Agenda

- Text Types
- Numeric Types
- Dates and Times
- XML
- JSON
- Boolean
- Bits
- Binary Data
- Network
- Arrays
- **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)

Composite Types

- Composite a new type from existing data types
- Same as ROW type

Composite Types

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

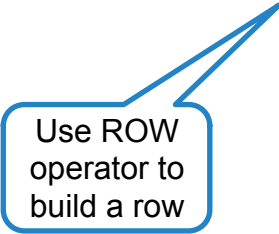
Looks like
a table

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

Used like any
other data type

Composite Types

```
INSERT INTO currency_history (data)
VALUES (ROW('EUR', 'USD', '2017-01-31', 1.0755)),
        (ROW('EUR', 'USD', '2017-01-30', 1.0630)),
        (ROW('EUR', 'USD', '2017-01-27', 1.0681)),
        (ROW('EUR', 'USD', '2017-01-26', 1.0700)),
        (ROW('EUR', 'USD', '2017-01-25', 1.0743)),
        (ROW('EUR', 'USD', '2017-01-24', 1.0748)),
        (ROW('EUR', 'USD', '2017-01-23', 1.0715));
```



Use ROW operator to build a row

Composite Types

```
SELECT * FROM currency_history;
```

id	data
1	(EUR,USD,2017-01-31,1.076)
2	(EUR,USD,2017-01-30,1.063)
3	(EUR,USD,2017-01-27,1.068)
4	(EUR,USD,2017-01-26,1.070)
5	(EUR,USD,2017-01-25,1.074)
6	(EUR,USD,2017-01-24,1.075)
7	(EUR,USD,2017-01-23,1.072)

(7 rows)

Returns a
row set

Composite Types

```
SELECT (data).cv_date, (data).cv_value FROM currency_history;
```

cv_date	cv_value
2017-01-31	1.076
2017-01-30	1.063
2017-01-27	1.068
2017-01-26	1.070
2017-01-25	1.074
2017-01-24	1.075
2017-01-23	1.072

(7 rows)

Specify row name
in round brackets

Enumerations

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

Question

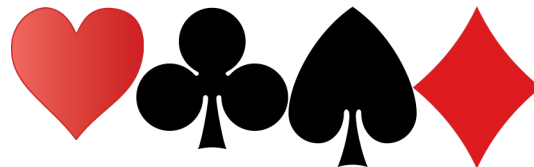
- ENUM is often used for gender
- How many different gender value do you know?



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';
```

id		color		winner
-----+-----+-----				
1		Hearts		Paul

What's missing?

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

THE END

