Standard-Datentypen

- **Numerisch**
- **Alphanumerisch**
- **Datum und Zeit**
- **Boolean**
- **Aufzählung**
- **Binär**

PostgreSQL-spezifische Datentypen

- **Money**
- **Bitstrings**
- **Netzwerkadressen**
- **Geometrische Datentypen**
- **Arrays**
- **XML**
- **UUID**
- **Zusammengesetzte Datentypen**
- **Tsearch**
- **Object Identifier**

Numerische Datentypen – Ganzzahlen

Smallint, int2	**-32768 bis +32767**
Integer, int4	**-2147483648 bis +2147483647**
Bigint, int8	**-9223372036854775808 bis +9223372036854775807**
serial	**1 bis 2147483647**
bigserial	**1 bis 9223372036854775807**

Es gibt kein signed / unsigned in PostgreSQL

serial und bigserial sind auto-increment-Typen, die im Hintergrund eine Sequenz anlegen mit Startwert = 1, Schrittweite = 1,

serial: Beispiel

```
create table seri (nr serial, inhalt text);
HINWEIS:  CREATE TABLE erstellt implizit eine Sequenz »seri_nr_seq« für
die »serial«-Spalte »seri.nr«
CREATE TABLE

                          Tabelle »public.seri«
 Spalte |   Typ   |                        Attribute
--------+---------+-----------------------------------------------------
 nr     | integer | not null Vorgabewert nextval('seri_nr_seq'::regclass)
 inhalt | text    |
```

Numerische Datentypen mit Nachkommastellen

numeric, decimal	**Bis zu 131072 Stellen vor dem DezimalTZ, bis zu 16383 Stellen nach dem DezimalTZ**
real	**6 Nachkommastellen (Rundungsfehler)**
double precision, float	**15 Nachkommastellen (Rundungsfehler)**

Syntax von numeric:

```
numeric(alleStellen[, Nachkommastellen])
numeric
```

Ohne Angabe von Stellen: Spalte mit unbestimmt vielen Stellen vor und hinter dem DezimalTZ, Werte werden nicht auf eine bestimmte Anzahl Stellen beschnitten

Alphanumerische Datentypen

character varying(n), varchar(n)	**Zeichenketten mit einer Länge von maximal n Zeichen Ohne n: beliebige Länge**
character(n), char(n)	**Zeichenkette mit einer Länge von maximal n Zeichen, leere Stellen werden mit Leerzeichen aufgefüllt. Ohne n: char(1)**
text	**Zeichenkette beliebiger Länge. (Maximal etwa 1 GB)**

text: ist nicht im SQL-Standard enthalten.

char: Leerzeichen am Ende werden beim Stringvergleich ignoriert und bei der Konversion in einen anderen Typ abgeschnitten.

Datum und Zeit

Name		Zeitraum	Einheit
timestamp [(*p*)] [without time zone]	Datum und Zeit ohne Zeitzone	Von 4713 v. Chr. bis 294276	1 Microsekunde / 14 Stellen
timestamp [(*p*)] with time zone, timestamptz	Datum und Zeit mit Zeitzone	Von 4713 v. Chr. bis 294276	1 Microsekunde / 14 Stellen
date	Datum: Jahr, Monat, Tag	Von 4713 v. Chr. bis 5874897	1 Tag
time [(*p*)] [without time zone]	Uhrzeit	Von 00:00:00 bis 24:00:00	1 Microsekunde / 14 Stellen
time [(*p*)] with time zone, timetz	Uhrzeit mit Zeitzone	00:00:00+1459 - 24:00:00-1459	1 Microsekunde / 14 Stellen
interval [*fields*] [(*p*)]	Zeitspanne	-178000000 Jahre +178000000 Jahre	1 Microsekunde / 14 Stellen

p: Anzahl der Nachkommastellen im Rückgabewert bei Sekunden

Boolean

bool, boolean	**Kann nur 2 Werte annehmen: wahr oder falsch (oder NULL)** **wahr: TRUE, 't', 'true', 'y', 'yes', 'on', '1'** **falsch: FALSE, 'f', 'false', 'n', 'no', 'off', '0'**

Aufzählungen

Aufzählungstypen muss man zuerst mit CREATE TYPE erzeugen, bevor man sie in einer Tabelle verwenden kann.

Beispiel:

```
CREATE TYPE wochentag AS ENUM ('MO', 'DI', 'MI', 'DO', 'FR', 'SA', 'SO');
create table plan (tag wochentag, was varchar(30));
insert into plan values ('SO', 'Weihnachtsmarkt');
```

Der binäre Datentyp bytea – binary strings

bytea	**Ein binärer String beliebiger Länge als Sequenz von Oktetten**

Unterschiede zu normalen Strings

- **Mit bytea kann man auch die binäre 0 und andere nicht druckbare Zeichen speichern.**
- **Operationen verarbeiten die 'rohen' Bytes und sind unabhängig von Zeichensätzen und locale-Einstellungen.**

Es gibt zwei Eingabe- und Ausgabeformate: escape (traditionelles Format) und hex (neu und default seit PG 9.0). Das Ausgabeformat wird vom Parameter bytea_output in der postgresql.conf bestimmt.

bytea entspricht von den Funktionen und Operatoren her den Typen BLOB oder Binary Large Object des SQL-Standards, hat aber ein anderes Eingabeformat.

Der Datentyp money

money	**-92233720368547758.08 bis +92233720368547758.07**

Die Ausgabe hängt von den locale-Einstellungen ab (lc_monetary), was zu Problemen beim Autausch von Daten zwischen Systemen mit unterschiedlichen locale-Einstellungen führen kann.

Zeitweise wurde dieser Datentyp in der Dokumentation als deprecated oder obsolet bezeichnet und es wurde empfohlen, stattdessen den Datentyp numeric mit 2 Nachkommastellen zu verwenden.

In den Release-Notes zur aktuellen Version 9.1:

- **Add support for dividing `money` by `money` (Andy Balholm)**
- **Add support for casting between `money` and `numeric` (Andy Balholm)**
- **Add support for casting from `int4` and `int8` to `money` (Joey Adams)**

Bitstrings

Zeichenketten, die nur die Werte 1 und 0 enthalten.

bit(n)	**Bitstring mit eine Länge von exakt n Zeichen**
bit varying(n)	**Bitstring mit maximal n Zeichen Länge**
bit	**Äquivalent zu bit(1)**
bit varying	**Bitstring unbegrenzter Länge**

Eingabeformat: VALUES (B'110', …)

Für Bitstrings gibt es die Operatoren:

- **Verkettung**
- **bitweises AND, OR, XOR, NOT**
- **shift left**
- **shift right**

PostgreSQL Datentypen

Beispiel Bitstrings: Boolesche Verknüpfungen direkt in der DB

```
create table bitvgl (zahl1 integer, zahl2 integer);
select zahl1, zahl1::bit(6), zahl2, zahl2::bit(6) from bitvgl;
```

```
 zahl1 | zahl1  | zahl2 | zahl2
-------+--------+-------+--------
     2 | 000010 |     5 | 000101
     3 | 000011 |     7 | 000111
```

```
...
```

```
select zahl1, zahl2, case when (zahl1::bit(6) & zahl2::bit(6))::int > 1
then 'treffer' else 'nicht enthalten' end as ergebnis from bitvgl;
```

```
 zahl1 | zahl2 |    ergebnis
-------+-------+-----------------
     2 |     5 | nicht enthalten
     3 |     7 | treffer
    25 |     9 | treffer
```

Alternativ:

```
case when (zahl1::bit(6) & zahl2::bit(6)) = 0::bit(6) then
```

Netzwerkadressen

Datentypen zur Speicherung von IPv4, Ipv6 und MAC-Adressen

cidr	**IPv4 und IPv6 Netzwerk-Spezifikation**
inet	**IPv4 und IPv6 Hostadressen und optional die Netzmaske**
macaddr	**Mac-Adressen**

Eingabeformate:

- **cidr und inet: adresse/y, IPv4 oder IPv6 Adressen, /y ist die Anzahl der Bits in der Netzmaske. Wenn die Netzmaske fehlt, wird es als ein einzelner Host interpretiert.**
  Unterschied: cidr akzeptiert keine „Einsen“ rechts von der Netzmaske.
- **macaddr: akzeptiert folgende Formate:** `'08:00:2b:01:02:03', '08-00-2b-01-02-03', '08002b:010203', '08002b-010203', '0800.2b01.0203', '08002b010203'`

Geometrische Datentypen (in der Ebene)

point	**Punkt in der Ebene**	**(x,y)**
line	**Gerade (Schnittpunkte der Geraden)**	**((x1,y1),(x2,y2))**
lseg	**Strecke (Endpunkte der Strecke)**	**((x1,y1),(x2,y2))**
box	**Rechteck (Gegenüberliegende Ecken)**	**((x1,y1),(x2,y2))**
path	**Geschlossener Pfad (ähnlich wie Polygon)** **Offener Pfad (mit eckigen Klammern)**	**((x1,y1),...)** **[(x1,y1),...]**
polygon	**Polygon**	**((x1,y1),...)**
circle	**Kreis (Mittelpunkt und Radius)**	**<(x,y),r>**

x und y sind Fließkommazahlen

Für Polygone und Geschlossene Pfade gibt es unterschiedliche Operationen.

PostgreSQL Datentypen

Beispiel: eine Datenbank mit Orten und ihren Koordinaten (nicht sphärisch!!)

```
select * from cities where name='Reutlingen';
    name     | longitude |  latitude  | country
-------------+-----------+------------+---------
 Reutlingen | 9.2166667 | 48.4833333 | DE


select name, country from cities where circle'((9.21,48.48),0.5)' @>
point(longitude,latitude);
           name           | country
--------------------------+---------
 Wendlingen am Neckar     | DE
 Weinstadt-Endersbach     | DE
 Weil der Stadt           | DE
 Waiblingen               | DE
 Tubingen                 | DE
 Stuttgart                | DE
 Sindelfingen             | DE
 Sigmaringen              | DE
 ...
```

Arrays – multidimensionale Arrays als Spaltenwerte

- **Es gibt keinen speziellen Datentyp „array“ in PostgreSQL. Arrays werden beim Erzeugen einer Tabelle generiert, indem man einen Datentyp mit eckigen Klammern angibt, zB integer[ ]**
- **Für Arrayelemente kann jeder Basistyp, Aufzählungstyp oder zusammengesetzte Datentyp verwendet werden.**
- **Alle Arrayelemente haben denselben Datentyp.**
- **Die Größe eines Arrays (Elemente und Dimensionen) ist unbestimmt.**
- **Bei multidimensionalen Arrays mussen die Ausdrücke bei der Eingabe den Dimensionen des Arrays entsprechen. Dimension in { }**
- **Die Definition von Arrays ist einfach, die INPUT-Syntax ist zwar logisch aufgebaut aber unübersichtlich und fehlerträchtig ;-((**

Beispiel: Arrays erzeugen und füllen

```
create table arrays (nr smallint, zahlen integer[], tiere varchar[]);

insert into arrays values (1,'{12,13,14,15}','{{Affe,Adler,Amsel},
{Bär,Barsch,Biene},{Dackel,Drossel,Dohle}}');

insert into arrays values (2,'{23,26,29}','{{Esel, Elster,Ente},{Fuchs,
Fasan,Fink}}');

insert into arrays values (3,'{30,34,35,36,37}','{{Hund,Hahn,Habicht},
{Igel, Iltis,Insekten}}');


select zahlen,tiere from arrays;

zahlen | {12,13,14,15}
tiere  | {{Affe,Adler,Amsel},{Bär,Barsch,Biene},{Dackel,Drossel,Dohle}}

zahlen | {23,26,29}
tiere  | {{Esel,Elster,Ente},{Fuchs,Fasan,Fink}}

zahlen | {30,34,35,36,37}
tiere  | {{Hund,Hahn,Habicht},{Igel,Iltis,Insekten}}
```

Beispiel: Einzelne Arrayelemente abfragen via Index (Startwert: 1)

```
select zahlen[2] from arrays;
 zahlen
--------
     13
     26
     34
```

```
select tiere[2][3] from arrays;
  tiere
----------
 Biene
 Fink
 Insekten
```

Bereichsabfragen – Array Slices (gedachte Rechtecke in einem Array)

```
select tiere[1:2][2:3] from arrays;
            tiere
---------------------------------
 {{Adler,Amsel},{Barsch,Biene}}
 {{Elster,Ente},{Fasan,Fink}}
 {{Hahn,Habicht},{Iltis,Insekten}}
```

Liefert die Elemente 1 bis 2 des Arrays der ersten Dimension und darin die Elemente 2 bis 3 des Subarrays der zweiten Dimension.

XML – Speicherung von XML-Daten

Im Unterschied zum Datentyp „text“ findet eine Prüfung auf Wohlgeformtheit der Eingabedaten statt und es gibt spezielle „typ-sichere“ Funktionen. Um die Funktionen benutzen zu können, muss PostgreSQL mit der Konfigurations-option - `-with-libxml` kompiliert worden sein.

Der Datentyp xml kann wohlgeformte xml-Dokumente (DOCUMENT) oder aber xml-Fragmente (CONTENT) speichern, aber nicht gegen eine DTD validieren.

```
create table xmldata (nr serial, daten xml);
insert into xmldata (daten) values ('<buch><titel>Der Herr der
Ringe</titel><autor>J. R. R. Tolkien</autor></buch>');
```

- **Zeichenketten können mit `XMLPARSE` zu xml-Daten transformiert werden und xml-Daten mit `XMLSERIALIZE` wieder zu Zeichenketten.**
- **xml-Daten können mit Xpath verarbeitet werden.**
- **Tabellen, Schemata, Datenbanken können als xml-Daten exportiert werden.**

UUID

Ein UUID (Universally Unique Identifier) ist eine Folge von gruppierten Hexadezimalzahlen (RFC 4122, ISO/IEC 9834-8:2005) im Format:

```
550e8400-e29b-41d4-a716-446655440000
```

In verteilten Anwendungen ist die Wahrscheinlichkeit zweier gleicher Bezeichner mit UUID geringer als mit Sequenzen, die nur innerhalb einer Datenbank eindeutige Werte garantieren.

Erlaubte Formate in PostgreSQL:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Composite Types – Zusammengesetzte Typen

Dieser Typ beschreibt die Struktur eines Datensatzes als eine Liste von Spaltennamen und deren Datentypen, allerdings sind keine Constraints möglich. PostgreSQL erlaubt die Verwendung von zusammengesetzten Datentypen fast (überall dort, wo auch Basistypen erlaubt sind.

Auch wenn eine Tabelle definiert wird, wird immer implizit ein zusammen-gesetzter Datentyp ROW erzeugt, der den Zeilentyp der Tabelle beschreibt.

Composite type definieren:

```
CREATE TYPE freund AS (name text, gebtag date);
```

Verwendung eines Composite types:

```
CREATE TABLE freunde (id integer, wer freund);
```

PostgreSQL Datentypen

Composite type: Daten einfügen

```
insert into freunde values (1, ROW('elli', '05.07.1980'));
insert into freunde values (2, ROW('paul', '23.05.1983'));
```

Daten abfragen

```
select * from freunde;
 id |        wer
----+-------------------
  1 | (elli,1980-07-05)
  2 | (paul,1983-05-23)
```

```
select (wer).name, (wer).gebtag from freunde;
 name |   gebtag
------+------------
 elli | 1980-07-05
 paul | 1983-05-23
```

Text Search Types

PostgreSQL definiert zwei Datentypen für die Volltextsuche.

tsvector	**Repräsentiert ein Dokument, das für die Textsuche aufbereitet wurde. Es ist eine sortierte Liste verschiedener „normalisierter“ Varianten von Worten eines Dokuments (lexeme) .**
tsquery	**Repräsentiert eine Volltext-Abfrage: sie enthält lexeme, nach denen gesucht werden soll. Lexeme können z.B. mit booleschen Operatoren AND &, OR | oder NOT ! verknüpft werden.**

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
    tsvector
   ----------------------------------------------------
    'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

Beispiel: ein tsvector aus einem String

OID – Object Identifiers

OIDs werden intern von PostgreSQL als PRIMARY KEYS für verschiedene Systemtabellen verwendet.

Wenn der Konfigurationsparameter default_with_oids (seit PG 8.1 OFF) gesetzt ist oder wenn beim CREATE TABLE die Option „with oids“ angegeben wird, erhalten auch die Datensätze in benutzerdefinierten Tabellen OIDs.

Um den OID für Datensätze auszugeben, kann man ihn einfach als Feldname in einer Abfrage verwenden:

```
SELECT oid,* FROM tabelle;
```

OIDs sollten nicht als PRIMARY KEY-Ersatz mißbraucht werden, denn:

- **OIDs sind als int4 implementiert und deshalb für große Datenbanken als eindeutiger Identifier nicht ausreichend.**
- **OIDs werden vom System generiert: Probleme bei Portierungen**