

Einführung in SQL

Datentypen

Prof. Dr. Jan Kirenz

HdM Stuttgart

Datentypen

- Bei der Erstellung von neuen Tabellen muss jeweils der Datentyp angegeben werden.
- Der Datentyp wird immer nach der Bezeichnung einer Spalte definiert. Hier ein Beispiel:

```
CREATE TABLE eagle_watch (  
  observed_date date,  
  eagles_seen integer  
);
```

- **Characters:** Jeder Buchstabe oder Symbole
- **Numbers:** ganze Zahlen und Brüche
- **Dates and Times:** Zeitbezogene Informationen

Datentypen

Characters

- **char(n)**: Eine Spalte bei welcher der Inhalt Zeichen mit einer fixen Länge n sind.
- Eine Spalte die mit char(20) definiert ist, kann bis zu 20 Zeichen in einer Reihe speichern.
- Falls weniger als 20 Zeichen eingegeben werden, fügt PostgreSQL automatische Leerzeichen ein.
- Dieser Datentyp kann auch als `character(n)` definiert werden - dies ist die übliche Vorgehensweise.

Datentypen

Characters

- **varchar(n)**: Mit n wird die maximale Länge definiert.
- Wenn weniger eingegeben wird, wird der Inhalt nicht mit Leerzeichen aufgefüllt. `character varying (n)`

Datentypen

Characters

- **text**: keine Limitierung - maximal 1 GB. Dies ist nicht Teil des SQL-Standards, wird aber bspw. bei Microsoft SQL und MySQL genutzt.

Datentypen

Characters

- In PostgreSQL existiert kein wesentlicher Unterschied zwischen den drei oben genannten Datentypen.
- Aufgrund der Effizienz und Flexibilität ist es in der Regel sinnvoll, **varchar** oder **text** zu nutzen.
- Eine Ausnahme stellen Eingaben dar, die immer eine bestimmte Länge aufweisen (wie bspw. PLZ).

Datentypen

- Schritt 1

```
CREATE TABLE char_data_types (  
  varchar_column varchar(10),  
  char_column char(10),  
  text_column text  
);
```

- Schritt 2

```
INSERT INTO char_data_types  
VALUES  
  ('abc', 'abc', 'abc'),  
  ('defghi', 'defghi', 'defghi');
```

- Schritt 3 nur bei lokaler Umgebung (nicht in der Cloud möglich)
Die Daten werden in den tmp-Ordner gespeichert (hier kann es eventuell zu Schwierigkeiten bei fehlenden Zugriffsrechten kommen)

```
COPY char_data_types TO '/tmp/typetest.txt'  
WITH (FORMAT CSV, HEADER, DELIMITER '|');
```

Dies ist der Ordnerpfad für Mac

Für Windows siehe S. 39

Datentypen

Zahlen (Numbers)

Im Gegensatz zu Characters (dort können auch Zahlen abgespeichert werden) können mit numerischen Zahlen mathematische Operationen durchgeführt werden.

- **Integers:** Ganze Zahlen
- **Fixed-point** und
- **floating-point**

Datentypen

Integers

- smallint: 2 bytes; integer: 4 bytes; bigint: 8 bytes
- Falls sehr große Datenwerte vorhanden sind (Datenformate größer als 2.1 Mrd), ist es empfehlenswert, mit bigint als Standard zu arbeiten.
- Ansonsten ist integer eine gute Wahl.
- Falls eine größere Zahl eingegeben wird, erscheint der Hinweis out of range.

Datentypen

Auto-incrementing integers

Diese Typen sind Spezialfälle der unterschiedlichen Zahlentypen. Der Index beginnt jeweils bei 1 und erhöht sich inkrementell. Hier ein Beispiel mit serial (smallserial und bigserial sind nicht in allen SQL-Datenbanken nutzbar)

- smallserial, 2 bytes, 1 bis 32767
- serial, 4 bytes, 1 bis 2.147.483.647
- bigserial, 8 bytes

```
CREATE TABLE people (  
  id serial,  
  person_name varchar(100)  
);
```

Datentypen

Dezimalzahlen

Fixed-Point Numbers

- `numeric(precision,scale)` (**arbitrary-precision type**)
 - **Precision** beschreibt die maximale Anzahl an Stellen links und rechts von dem Komma (US: decimal point).
 - **Scale** beschreibt die maximale Anzahl an Stellen rechts von dem Komma (US: right of the decimal point).

Datentypen

Dezimalzahlen

Fixed-Point Numbers

- `Decimal(precision,scale)`
- Alternativ zu `numeric` kann auch `decimal` genutzt werden.
- Beide sind Teil des ANSI SQL Standards.
- Falls der **Scale** Wert nicht definiert wird, wird dieser automatisch auf 0 gesetzt (d.h. er wird zu einem integer).
- Falls **Precision** und **Scale** nicht definiert werden, wird die höchst mögliche Anzahl genutzt.

Datentypen

Floating-Point Types

- Dieser Typ wird auch **variable-precision-type** genannt.
- Die Datenbank speichert dabei die Zahl als Nummer und einen Exponenten, der angibt, an welcher Stelle sich der Dezimalpunkt befindet.
- Es muss dabei **kein** `precision` **und** `scale` **angegeben** werden.
 - **real**, insgesamt 6 Zahlen vor und nach dem Komma.
 - **double precision**, insgesamt 15 Zahlen vor und nach dem Komma.
- Der Unterschied der beiden besteht also darin, wieviel Daten sie speichern können.

Datentypen

Vergleich der numerischen Typen

- Wir erzeugen eine Tabelle um die unterschiedlichen Formate zu vergleichen:

```
CREATE TABLE number_data_types (  
  numeric_column numeric(20,5),  
  real_column real,  
  double_column double precision  
);
```

Datentypen

```
INSERT INTO number_data_types  
VALUES  
  (.7, .7, .7),  
  (2.13579, 2.13579, 2.13579),  
  (2.1357987654, 2.1357987654, 2.1357987654);
```

Datentypen

```
SELECT *  
FROM number_data_types;
```

- Ausgabe einer standard SQL-Abfrage (bspw. mit pgAdmin)

numeric_column	real_column	double_column
0.70000	0.7	0.7
2.13579	2.13579	2.13579
2.13580	2.1358	2.1357987654

- Ausgabe des R-SQL-Pakets (darin existieren nur numeric Werte)

numeric_column	real_column	double_column
0.70000	0.700000	0.700000
2.13579	2.135790	2.135790
2.13580	2.135799	2.135799

Datentypen

```
SELECT
  numeric_column * 10000000 AS "Fixed",
  real_column * 10000000 AS "Float"
FROM number_data_types
WHERE numeric_column = .7;
```

- Ausgabe einer standard SQL-Abfrage (bspw. mit pgAdmin)

Fixed	Float
7000000.00000	6999999.88079071

- Aus diesem Grund werden "Floating-Types" als nicht exakt bezeichnet
- Ausgabe des R-SQL-Pakets

Fixed	Float
7e+06	7e+06

Datentypen

Datum und Uhrzeit

- PostgreSQL unterstützt die wichtigsten Zeitformen
- **timestamp** speichert Datum und Zeit
- **date Records** speichert nur das Datum.
- **time Records** nur die Uhrzeit.

Datentypen

```
CREATE TABLE date_time_types (  
    timestamp_column timestamp with time zone,  
    interval_column interval  
);
```

```
INSERT INTO date_time_types  
VALUES  
    ('2018-12-31 01:00 EST','2 days'),  
    ('2018-12-31 01:00 PST','1 month'),  
    ('2018-12-31 01:00 Australia/Melbourne','1 century'),  
    (now(),'1 week');
```

Datentypen

```
SELECT *  
FROM date_time_types;
```

timestamp_column	interval_column
2018-12-31 07:00:00	2 days
2018-12-31 10:00:00	1 mon
2018-12-30 15:00:00	100 years
2021-01-03 18:07:00	7 days

Datentypen

Interval Data Type

```
SELECT
  timestamp_column,
  interval_column,
  timestamp_column - interval_column AS new_date
FROM date_time_types;
```

timestamp_column	interval_column	new_date
2018-12-31 07:00:00	2 days	2018-12-29 07:00:00
2018-12-31 10:00:00	1 mon	2018-11-30 10:00:00
2018-12-30 15:00:00	100 years	1918-12-30 15:00:00
2021-01-03 18:07:00	7 days	2020-12-27 18:07:00

Datentransformationen mit CAST()

- Die CAST-Funktion kann nur angewendet werden, wenn die originären Werte in das gewünschte Format überführt werden können.
- Beispielsweise können Zahlen als Text gespeichert werden, nicht jedoch Text als Zahlen.
- Hier werden die ersten zehn Ziffern der Spalte in ein neues Format umgewandelt

Datentransformationen mit CAST()

```
SELECT timestamp_column,  
       CAST(timestamp_column AS varchar(10))  
FROM date_time_types;
```

timestamp_column	timestamp_column..2
2018-12-31 07:00:00	2018-12-31
2018-12-31 10:00:00	2018-12-31
2018-12-30 15:00:00	2018-12-30
2021-01-03 18:07:00	2021-01-03

Datentransformationen mit CAST()

```
SELECT numeric_column,  
       CAST(numeric_column AS integer),  
       CAST(numeric_column AS varchar(6))  
FROM number_data_types;
```

numeric_column	numeric_column..2	numeric_column..3
0.70000	1	0.7000
2.13579	2	2.1357
2.13580	2	2.1358

Datentransformationen mit CAST()

CAST Shortcut

- Der Befehl CAST kann (nur!) in PostgreSQL mit :: abgekürzt werden

```
SELECT timestamp_column, CAST(timestamp_column AS varchar(10))  
FROM date_time_types;
```

timestamp_column	timestamp_column..2
2018-12-31 07:00:00	2018-12-31
2018-12-31 10:00:00	2018-12-31
2018-12-30 15:00:00	2018-12-30
2021-01-03 18:07:00	2021-01-03

Datentransformationen mit CAST()

CAST Shortcut

```
SELECT timestamp_column::varchar(10)  
FROM date_time_types;
```

timestamp_column
2018-12-31
2018-12-31
2018-12-30
2021-01-03

Datenimport und Datenexport

- In PostgreSQL kann der Befehl COPY für das Einlesen und Exportieren verwendet werden
- Überblick über [Datenkonvertierungen in PostgreSQL](#)
- Vorgehensweise für den Import von Daten:
 1. Quelldaten als CSV bereitstellen
 2. Eine Tabelle für die Datenspeicherung vorbereiten
 3. Den COPY-Befehl für den Import der Daten schreiben
- Beispiel einer CSV-Datei:

```
FIRSTNAME,LASTNAME,STREET,CITY,STATE,PHONE  
John,Doe,"123 Main St., Apartment 200",Hyde Park,NY,845-555-1212
```

Datenimport und Datenexport

CSV Daten

- Falls in den CSV-Daten Kommas verwendet werden, müssen die entsprechenden Werte in den Spalten mit Anführungszeichen versehen werden.

John,Doe,"123 Main St., Apartment 200",Hyde Park,NY,845-555-1212

Datenimport mit COPY

- Beispielhafter Syntax

WINDOWS:

```
COPY table_name  
FROM 'C:\YourDirectory\your_file.csv'  
WITH (FORMAT CSV, HEADER);
```

MAC oder Linux

```
COPY table_name  
FROM 'Users/YourDirectory/your_file.csv'  
WITH (FORMAT CSV, HEADER);
```

Import von Daten am Beispiel von Zensusdaten (USA 2010)

Vollständiger Code ist in [GitHub](#)
verfügbar: [sql_us_counties.pgsql](#)

- Für mehr Informationen zu den Daten, siehe das [Data dictionary](#)

```
CREATE TABLE us_counties_2010 (  
  geo_name varchar(90),           -- Name of the geography  
  state_us_abbreviation varchar(2), -- State/U.S. abbreviation  
  summary_level varchar(3),       -- Summary Level  
  region smallint,                -- Region  
  division smallint,              -- Division  
  state_fips varchar(2),           -- State FIPS code  
  county_fips varchar(3),         -- County code  
  area_land bigint,               -- Area (Land) in square meters  
  area_water bigint,              -- Area (Water) in square meters  
  population_count_100_percent integer, -- Population count (100%)  
  housing_unit_count_100_percent integer, -- Housing Unit count (100%)  
  internal_point_lat numeric(10,7), -- Internal point (latitude)  
  internal_point_lon numeric(10,7), -- Internal point (longitude)  
  
  -- This section is referred to as P1. Race:  
  p0010001 integer, -- Total population  
  p0010002 integer, -- Population of one race:  
  p0010003 integer, -- White Alone  
  p0010004 integer, -- Black or African American alone  
  p0010005 integer, -- American Indian and Alaska Native alone  
  p0010006 integer, -- Asian alone  
  p0010007 integer, -- Native Hawaiian and Other Pacific Islander alone
```

Import von Daten am Beispiel von Zensusdaten (USA 2010)

- Benötigte CSV: [us_counties_2010.csv](#)

```
COPY us_counties_2010  
FROM '/tmp/us_counties_2010.csv'  
WITH (FORMAT CSV, HEADER);
```

```
SELECT *  
FROM us_counties_2010  
LIMIT 40;
```

Show entries

Search:

	geo_name ♦	state_us_abbreviation ♦	summary_level ♦	region ♦	division
1	Autauga County	AL	050	3	6
2	Baldwin County	AL	050	3	6
3	Barbour County	AL	050	3	6
4	Bibb County	AL	050	3	6
5	Blount County	AL	050	3	6
6	Bullock County	AL	050	3	6

Showing 1 to 6 of 40 entries

Previous

1

2

3

4

5

6

7

Next

Zensusdaten (USA 2010)

```
SELECT geo_name, state_us_abbreviation, area_land  
FROM us_counties_2010  
ORDER BY area_land DESC  
LIMIT 3;
```

geo_name	state_us_abbreviation	area_land
Yukon-Koyukuk Census Area	AK	376855656455
North Slope Borough	AK	229720054439
Bethel Census Area	AK	105075822708

Zensusdaten (USA 2010)

```
SELECT geo_name, state_us_abbreviation, internal_point_lon
FROM us_counties_2010
ORDER BY internal_point_lon DESC
LIMIT 5;
```

geo_name	state_us_abbreviation	internal_point_lon
Aleutians West Census Area	AK	178.33881
Washington County	ME	-67.60935
Hancock County	ME	-68.37070
Aroostook County	ME	-68.64941
Penobscot County	ME	-68.65749

Import einer Datenteilmenge (Supervisor salaries)

- **CREATE TABLE:**

```
CREATE TABLE supervisor_salaries (  
  town varchar(30),  
  county varchar(30),  
  supervisor varchar(30),  
  start_date date,  
  salary money,  
  benefits money  
);
```

Import einer Datenteilmenge (Supervisor salaries)

- Daten sind hier verfügbar: [supervisor_salaries.csv](#)

```
COPY supervisor_salaries (town, supervisor, salary)
FROM '/tmp/supervisor_salaries.csv'
WITH (FORMAT CSV, HEADER);
```

```
-- Check the data
SELECT *
FROM supervisor_salaries
LIMIT 2;
```

town	county	supervisor	start_date	salary	benefits
Anytown	NA	Jones	NA	0	NA
Bumblyburg	NA	Baker	NA	0	NA

Vielen Dank!

Prof. Dr. Jan Kirenz

HdM Stuttgart
Nobelstraße 10
70569 Stuttgart

