
# PostgreSQL


## Bereichstypen: Was sind sie und was können sie?

Bereichstypen sind eine sehr praktische Besonderheit von PostgreSQL

Holger@Jakobs.com – http://plausibolo.de

FrOSCon 2016

# Vorstellung Holger Jakobs

- studierter Wirtschaftsinformatiker
- jahrzehntelange Erfahrung in der Ausbildung von Software-Entwicklern
- seit 2013 freiberuflich als Berater und Trainer
- Lehrauftrag an der Fachhochschule der Wirtschaft für Betriebssysteme
- LPIC-2-zertifiziert und PRINCE2 Practitioner
- Fan von PostgreSQL, Tcl/Tk, Fossil, …
- Motto: KISS – keep it simple & stupid

# PostgreSQL auf einen Blick

- Andere DBMS mögen populärer sein, aber es gibt kein fortschrittlicheres oder standard-näheres DBMS.
- NoSQL Datenbanken sind **keine** Alternative zur Speicherung strukturierter Daten.
- Umgekehrt macht PostgreSQL sogar bei der Speicherung nicht einheitlich strukturierter Daten eine gute Figur, z. B. bei JSON- oder key-value-Daten.
- Größter Anwender in Deutschland: zalando

# Sinn und Zweck von RDBMS

- Bloße Speicherung von (viel) Daten? **NEIN!**
- ACID: **a**tomicity, **c**onsistency, **i**solatability, **d**urability
- Wesentlich hierbei ist die Konsistenz, d. h. dass Daten nicht widersprüchlich oder ungültig sein dürfen.
- Das **muss** das RDBMS sicher stellen und **nicht** die Anwendung! (sorry, liebe PHP+MySQL-Entwickler!)
- Konsistenzbedingungen wollen **deklariert** und **nicht** prozedural/objektorientiert ausprogrammiert werden!

# Was hilft bei der Konsistenzprüfung?

- ganz wichtig: passende **Datentypen**!
- Eine Datumsspalte darf keine ungültigen Datumswerte wie `'2016-00-00'` oder `'2016-04-31'` annehmen, sondern **muss** einen Fehler melden.
- Der Versuch, 30 Zeichen in einer Spalte vom Typ `VARCHAR(20)` abzulegen, **muss** einen Fehler melden – und darf weder 30 Zeichen speichern noch 10 Zeichen abschneiden.

# Was hilft bei der Konsistenzprüfung?

- Der Versuch, eine Spalte vom Typ `VARCHAR(20)` auf 10 Zeichen zu kürzen, wenn sie auch Zeichenketten mit mehr als 10 Zeichen enthält, **muss** einen Fehler melden.
- Der Versuch, den Wert -5 in der Spalte `betrag` vom Typ `INTEGER` abzulegen, welche eine Bedingung `CHECK(betrag >= 0)` trägt, **muss** einen Fehler melden.
- Der Versuch, einen Kunden zu löschen, zu dem es Bestellungen gibt, **muss** einen Fehler melden.

# Was hilft bei der Konsistenzprüfung?

- Der Versuch, eine Spalte *ohne* default-Wert, aber *mit* dem Zusatz `NOT NULL` beim Eintragen einer neuen Zeile nicht zu bedienen, **muss** einen Fehler melden.
- **Alle** diese Prüfungen müssen auch dann aktiv sein, wenn jemand ein interaktives Tool verwendet – und nicht die programmierte Anwendung.
- Schön ist es, wenn auch so etwas wie IP- und MAC-Adressen auf Korrektheit geprüft werden.

# Bereichstypen in PostgreSQL

- Bereichswerte kommen häufig vor:
  - Auslieferungsgebühren bei Möbelhändlern
  - Punkteschema für Klausurnoten
  - Mitgliedschaft-Dauer und Amtszeiten
  - Mietdauer von Autos
  - Versicherungsprämie in Abhängigkeit von Preisbereich der zu versichernden Güter
  - Zeitraum für Rabattaktionen
  - Hotelbuchungen
  - Konferenzraumbelegungen

# Traditionelle Lösung

- Speicherung des von- und des bis-Werts in zwei Spalten
- Ob ein Wert im Bereich liegt, lässt sich relativ leicht prüfen.
- Ob sich zwei Bereiche überschneiden, ist schon schwieriger.
- Eine Bedingung für Nicht-Überschneidung zu formulieren, ist sehr aufwendig und fehlerträchtig.

# PostgreSQL-Bereichstypen

Es gibt vordefinierte Bereichstypen mit folgenden Basistypen:

- `int4range` – INTEGER
- `int8range` – BIGINT
- `numrange` – NUMERIC
- `daterange` – DATE
- `tsrange` – TIMESTAMP
- `tstzrange` – TIMESTAMP WITH TIMEZONE

# Umgang mit Bereichstypen

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES (1108,
            '[2010-01-01 14:30, 2010-01-01 15:30)');

SELECT int4range(10, 20) @> 3;    -- enthalten oder nicht?

SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);
-- Überlappung vorhanden?

SELECT upper(int8range(15, 25));  -- obere Grenze ermitteln

SELECT int4range(10, 20) * int4range(15, 25); -- Schnittmenge

SELECT isempty(numrange(1, 5));  -- leer?
```

# Bereichstyp selbst definieren

```
CREATE TYPE inetrange AS RANGE (
  SUBTYPE = inet
);


SELECT '192.168.1.8'::inet <@
    inetrange('192.168.1.1',
              '192.168.1.10');
  => TRUE


SELECT '192.168.1.20'::inet <@
    inetrange('192.168.1.1',
              '192.168.1.10');
  => FALSE
```

# Erzeugung von Bereichswerten

```
SELECT numrange(1.0, 14.0, '(]');

SELECT numrange(1.0, 14.0);
-- als wäre '[)' eingetragen

SELECT numrange(NULL, 2.2);
-- unten offener Bereich
```

# Konsistenzprüfungen mit Bereichen

```
CREATE EXTENSION btree_gist;
```

lädt die notwendige Erweiterung, benötigt

```
# apt install postgresql-contrib-9.5
```

```
CREATE TABLE reservation (
  res_id  INTEGER REFERENCES ressourcen,
  zeitspanne tsrange,
  EXCLUDE USING GIST
    (res_id WITH =, zeitspanne WITH &&)
);
```

verhindert Doppelreservierungen derselben Ressource.

# Besonderes Beispiel

**Aufgabe:** Erstellung eines Wochenstundenplans für eine Bildungseinrichtung (Schule, Hochschule)

- Es gibt Räume, Dozenten und zu planende Vorlesungen.
- Doppelbuchungen von Räumen und von Dozenten muss vermieden werden.
- Zeitangaben der Vorlesungen bestehen aus Wochentag, Beginnzeit und Endezeit.

# Praxis!!

Das lösen wir jetzt einfach mal praktisch.

# Wochenstundenplan-Erstellung für Dozenten und Räume mit PostgreSQL

Holger Jakobs – holger@jakobs.com

2016-08-19

## Inhaltsverzeichnis

**1 Aufgabenstellung** **1**

**2 Umsetzung** **2**
2.1 Die einfachen Tabellen . . . . . 2
2.2 Die Vorlesungstabelle . . . . . 2
2.2.1 Bereichstypen . . . . . 2
2.2.2 Wochentag . . . . . 3
2.2.3 Tabellenerzeugung . . . . . 4
2.2.4 Sicht und Regeln . . . . . 5
2.3 Testdateneingabe . . . . . 6
2.4 Abfragen . . . . . 7

# 1 Aufgabenstellung

Es soll mithilfe einer Datenbank die Erstellung eines Wochenstundenplans für Räume und Dozenten – z. B. an einer Hochschule – organisiert werden. Bislang war so etwas immer extrem kompliziert und hat viel Programmierung verlangt – sei es im Frontend (schlecht gemacht) oder im Backend (schon besser). Aber wie wäre es, wenn ein Datenbanksystem die notwendigen Integritätsprüfungen, also die Verhinderung von Doppelbelegungen von Räumen und Doppelverplanung von Dozenten, allein durch die Einrichtung von *statischen Prüfungen* ermöglicht – also *ohne Trigger*?

Der zu erstellende Plan ist ein Wochenstundenplan, also kein Plan für konkrete Zeiträume mit zwei `TIMESTAMP`s als Anfang und Ende, sondern ein Plan, der sich jede Woche im kommenden Semester wiederholen soll. Anfang und Ende der Zeiträume sind also jeweils eine Kombination aus einer Wochentagsangabe und einer Uhrzeit (`TIME`). Das macht die Angelegenheit entsprechend schwieriger, denn für diese Kombination gibt es keinen vordefinierten Datentypen.

Mit PostgreSQL haben wir das leistungsfähigste Datenbanksystem überhaupt an der Hand, so dass man hier durchaus wagen kann, einen Versuch zu starten, das umzusetzen. Alle anderen Datenbanksysteme – seien es freie oder kommerzielle Systeme – haben da keine Chance, denn der zugrunde liegende Datentyp für einen Zeitabschnitt, ein sogenannter `RANGE TYPE`, fehlt im Standard und in allen anderen Datenbanksystemen. Höchstens DB2, welches sogar schon die temporalen Typen von SQL:2011 bietet, könnte hier evtl. eine Lösung anbieten.

# 2 Umsetzung

## 2.1 Die einfachen Tabellen

Die Tabellen für die Räume und die Dozenten sollen bewusst ganz einfach und für die Praxis unvollständig gehalten werden.

```
CREATE TABLE raum (
  rnr          SERIAL PRIMARY KEY,
  anzplaetze   INTEGER NOT NULL CHECK (anzplaetze > 0)
);

CREATE TABLE dozent (
  dnr          SERIAL PRIMARY KEY,
  name         VARCHAR(30) NOT NULL
);
```

## 2.2 Die Vorlesungstabelle

### 2.2.1 Bereichstypen

Es gibt in PostgreSQL sogenannte „Bereichstypen“ (`RANGE TYPES`), von denen einige vordefiniert und andere nach Bedarf hinzugefügt werden können. Die vordefinierten Bereichstypen sind:

- `int4range` – `INTEGER`-Bereich
- `int8range` – `BIGINT`-Bereich
- `numrange` – `NUMERIC`-Bereich
- `daterange` – `DATE`-Bereich
- `tsrange` – `TIMESTAMP`-Bereich
- `tstzrange` – `TIMESTAMP WITH TIMEZONE`-Bereich

Weil die Planer einen Wochenstundenplan erstellen wollen, kann für die einzelne Vorlesung kein `TIMESTAMP`-Typ verwendet werden, denn das wäre dann immer ein konkretes Datum, aber es geht ja nur um Zeiträume bezogen auf eine (beliebige) Woche. Also bietet es sich an, für „Montag 08:00 bis 09:00“ zwei Spalten zu verwenden. Die erste legt den Wochentag fest, die zweite den Zeitraum.

Aber halt! Gibt es denn einen Bereichstypen für `TIME`? Nein, der ist leider nicht vordefiniert. Also müssen wir hier selbst tätig werden und diesen definieren.

Um einen neuen, eigenen Bereichstypen definieren zu können, benötigen wir zunächst einmal eine Funktion, um die Differenz zweier Werte auszurechnen und diese in Form eines Wertes vom Typ `DOUBLE PRECISION` zurückliefert. Schließlich muss das Datenbanksystem feststellen können, welcher von zwei Werten kleiner ist (denn in Bereichen muss die Untergrenze kleiner oder gleich der Obergrenze sein) und mithilfe dieser Funktion auch prüfen können, ob sich Bereiche überlappen.

Leider fehlt auch eine solche Funktion und muss selbst erstellt werden. Die Funktion muss zudem „immutable“ sein, d. h. es muss garantiert werden, dass diese Funktion nichts an den Daten der Datenbank verändert und auch keine anderen Informationen verwendet als ihre Parameterliste. Dies erreicht man schlicht durch den Zusatz genau dieses Wortes, denn dadurch spricht man eine entsprechende Zusicherung aus. Die Funktion an sich ist so einfach, dass man dafür nicht einmal PL/pgSQL bemühen muss, sondern sie direkt in SQL implementieren kann. So kann diese Funktion aussehen:

```
CREATE FUNCTION timediff (a time, b time) RETURNS DOUBLE PRECISION AS $$
  SELECT EXTRACT(EPOCH FROM (a - b));
$$ LANGUAGE SQL IMMUTABLE;
```

Darauf basierend kann nun der neue `timerange`-Typ definiert werden. Es kann gut sein, dass er in zukünftigen Versionen von PostgreSQL vordefiniert sein wird.

```
CREATE TYPE timerange AS RANGE (
    subtype = TIME,
    subtype_diff = timediff
);
```

### 2.2.2 Wochentag

Den Wochentag könnte man als Aufzähltypen (`ENUM`) in der Datenbank ablegen, weil PostgreSQL auch Aufzähltypen unterstützt. Aber wegen der notwendigen Funktionen, die für den Typen zur Verfügung stehen müssen, bereitet das leider Schwierigkeiten, so dass wir hier auf einen einfachen `INTEGER` zurückgreifen (`SMALLINT` täte es natürlich auch).

Damit aber nach außen trotzdem mit symbolischen Werten für die Wochentage gearbeitet werden kann, wird eine entsprechende Sicht (`VIEW`) erstellt. Diese versteckt die „numerische“ Implementation vollständig.

Wir richten den Wochentagstyp `wo_t` als Aufzähltyp ein und fügen je eine Konversionsfunktion von `wo_t` nach `INTEGER` und zurück hinzu. Auch diese Funktionen werden in reinem SQL geschrieben.

```
CREATE TYPE wo_t AS ENUM ('mo', 'di', 'mi', 'do', 'fr');

CREATE FUNCTION int2wo_t(tag INTEGER) RETURNS wo_t AS $$
  SELECT CASE
    WHEN tag = 1 THEN 'mo'::wo_t
    WHEN tag = 2 THEN 'di'::wo_t
    WHEN tag = 3 THEN 'mi'::wo_t
    WHEN tag = 4 THEN 'do'::wo_t
    WHEN tag = 5 THEN 'fr'::wo_t
    ELSE NULL
  END;
$$ LANGUAGE SQL;

create function wo_t2int(tag wo_t) RETURNS INTEGER AS $$
  SELECT CASE
    WHEN tag = 'mo'::wo_t THEN 1
    WHEN tag = 'di'::wo_t THEN 2
    WHEN tag = 'mi'::wo_t THEN 3
    WHEN tag = 'do'::wo_t THEN 4
    WHEN tag = 'fr'::wo_t THEN 5
    ELSE NULL
  END;
$$ LANGUAGE SQL;
```

### 2.2.3 Tabellenerzeugung

Nun haben wir alles bereit, um die Tabelle für die Vorlesungen zu generieren. Allerdings nennen wir sie `vorlesung_intern`, weil sie nach außen hin nicht verwendet wird.

```
CREATE TABLE vorlesung_intern (
  dnr         INTEGER NOT NULL REFERENCES dozent,
  rnr         INTEGER NOT NULL REFERENCES raum,
  thema       VARCHAR(30) NOT NULL,
  tag         INTEGER NOT NULL check (tag BETWEEN 1 AND 5),
  zeit        timerange NOT NULL,
  PRIMARY KEY (rnr, tag, zeit),
  UNIQUE (dnr, tag, zeit),
  CONSTRAINT "Doppelbelegung Raum"
  EXCLUDE USING GIST (rnr WITH =, tag WITH =, zeit WITH &&),
  CONSTRAINT "Doppelverplanung Dozent"
  EXCLUDE USING GIST (dnr WITH =, tag WITH =, zeit WITH &&)
);
```

Neben den offensichtlichen Spalten für Dozenten- und Raumnummer, welche Fremdschlüssel auf die entsprechenden Tabellen darstellen, gibt es noch das Thema – eine einfache Zeichenkette – und dann die Angaben für die Zeitplanung. Das ist eine Tagnummer im Bereich 1 bis 5, denn am Wochenende gibt es keine Vorlesungen, und ein Zeitbereich vom oben selbst erstellten Type `timerange`.

Primärschlüssel und eindeutige Spaltenkombination sind Raumnummer + Tag + Zeit oder Dozentennummer + Tag + Zeit – welches man als Primärschlüssel und welches nur auf `UNIQUE` setzt, kann man sich aussuchen. Auf beide könnte man ggf. Fremdschlüssel aus anderen Tabellen setzen, z. B. für die Zuordnung von geplanten Prüfungen.

Zur Sicherstellung, dass weder Räume doppelt belegt noch Dozenten doppelt eingeplant werden können, werden die `EXCLUDE`-Constraints verwendet. Ihr Einsatz verlangt eine Extension, die mittels `CREATE EXTENSION BTREE_GIST;` installiert wird, welche bei Debian im Paket `postgresql-contrib-`*`versionsnummer`* enthalten ist.

### 2.2.4 Sicht und Regeln

Nun wollen wir aber eine „Tabelle“ haben, mit der wir über die Wochentagsnamen statt -nummern kommunizieren. Also erzeugen wir eine entsprechende Sicht:

```
CREATE VIEW vorlesung AS
SELECT dnr, rnr, thema, int2wo_t(tag) AS tag, zeit
FROM vorlesung_intern;
```

Dummerweise ist diese Sicht nicht updatefähig, weil sie eine Funktion enthält. Einfache Sichten ohne Funktionen und ohne Joins können ggf. updatefähig sein, aber hier weiß das Datenbanksystem nicht, wie es eine symbolische Angabe in die entsprechende Wochentagsnummer zurückverwandeln kann. Also muss man für alle Schreiboperationen, also `INSERT`, `UPDATE` und `DELETE` je eine Regel einrichten, in der die passende Funktion aufgerufen wird.

```
CREATE RULE vorlesung_ins AS ON INSERT TO vorlesung
DO INSTEAD
  INSERT INTO vorlesung_intern
  VALUES (new.dnr, new.rnr, new.thema, wo_t2int(new.tag), new.zeit);

CREATE RULE vorlesung_del AS ON DELETE TO vorlesung
DO INSTEAD
  DELETE FROM vorlesung_intern
  WHERE (rnr = old.rnr OR dnr = old.dnr)
    AND tag = wo_t2int(old.tag) AND zeit = old.zeit;

CREATE RULE vorlesung_upd AS ON UPDATE TO vorlesung
DO INSTEAD
  UPDATE vorlesung_intern SET dnr = new.dnr, rnr = new.rnr,
    thema = new.thema, tag = wo_t2int(new.tag), zeit = new.zeit
```

```
  WHERE (rnr = old.rnr OR dnr = old.dnr)
    AND tag = wo_t2int(old.tag) AND zeit = old.zeit;
```

## 2.3 Testdateneingabe

Wenn jetzt alles stimmt, können wir Testdaten eingeben. Das Einfügen von Dozenten und Räume ist trivial.

```
INSERT INTO raum (anzplaetze) VALUES (30);
INSERT INTO raum (anzplaetze) VALUES (40);
INSERT INTO raum (anzplaetze) VALUES (35);

INSERT INTO dozent (name) VALUES ('Meier');
INSERT INTO dozent (name) VALUES ('Müller');
INSERT INTO dozent (name) VALUES ('Schmitz');
```

Das Einfügen von Vorlesungen testen wir natürlich gleich unter Verwendung der Sicht.

```
INSERT INTO vorlesung VALUES (1, 1, 'SQL', 'mo', '[08:00,09:30)');
INSERT INTO vorlesung VALUES (1, 2, 'Normalisierung', 'mo', '[09:45,11:15)');
```

Diese sind gut gegangen. Fragen wir mal ab, was eingetragen wurde:

```
SELECT * FROM vorlesung;
 dnr | rnr |     thema      | tag |        zeit
-----+-----+----------------+-----+---------------------
   1 |   1 | SQL            | mo  | [08:00:00,09:30:00)
   1 |   2 | Normalisierung | mo  | [09:45:00,11:15:00)
```

Also probieren wir mal je einen Eintrag aus, bei dem einmal der Dozent und einmal der Raum doppelt verplant wurde, d. h. die Zeiträume überlappen sich, was ohne dieses `EXCLUDES`-Feature von PostgreSQl fast unmöglich zu realisieren wäre.

```
INSERT INTO vorlesung VALUES (1, 3, 'Datenpflege', 'mo', '[09:00,10:30)');
INSERT INTO vorlesung VALUES (3, 1, 'JDBC', 'mo', '[09:00,10:30)');
```

Um herauszufinden, welche Vorlesung mit dem Termin des Dozenten Nr. 1 am Montag zwischen 09:00 und 10:30 kollidiert, fragen wir das einfach ab.

```
SELECT * FROM vorlesung
WHERE dnr = 1 AND tag =  'mo' AND zeit && '[09:00,10:30)';
```

Aha, dieser Eintrag würde sogar mit mehreren Vorlesungen kollidieren, weil das ein ziemlich langer Termin ist.

## 2.4 Abfragen

Vielleicht ist es auch interessant zu sehen, wie die Auslastung der Räume und Dozenten ist. Das kann mittels `SELECT`-Abfragen geschehen, die selbstverständlich auch als `VIEW`s abgelegt werden können.

Um die verplante Unterrichtszeit pro Dozent und Wochentag zu ermitteln, verwendet man die folgende Abfrage.

```
SELECT dnr, tag, sum (UPPER(zeit)-LOWER(zeit)) AS stunden
FROM vorlesung
GROUP BY dnr, tag;
```

Wenn der Name des Dozenten auch dabei stehen soll, dann `JOIN`t man die Dozententabelle dazu. Diese geschieht hier mittels `NATURAL JOIN`, denn auf diese Weise werden die Tabellen über alle gleichnamigen Spalten miteinander verknüpft – in diesem Falle also über die Spalte `dnr`.

```
SELECT dnr, name, tag, sum (UPPER(zeit)-LOWER(zeit)) AS stunden
FROM vorlesung NATURAL JOIN dozent
GROUP BY dnr, name, tag;
```

Vielleicht will man auch nur die Gesamtsumme der Wochenstunden wissen und die Anzahl Tage, an denen Einsätze stattfinden.

```
SELECT dnr, name, COUNT(DISTINCT tag) AS anzahltage,
  SUM (UPPER(zeit)-LOWER(zeit)) AS stunden
FROM vorlesung NATURAL JOIN dozent
GROUP BY dnr, name;
```

Oder die Einsatztage sollen neben der Wochenstundenanzahl aufgeführt werden.

```
SELECT dnr, name, array_agg(distinct tag) AS einsatztage,
  TO_CHAR(SUM(UPPER(zeit)-LOWER(zeit)),'HH24:MI') AS stunden
FROM vorlesung NATURAL JOIN dozent
GROUP BY dnr, name;
```

```
 dnr |  name  |   einsatztage    | stunden
-----+--------+------------------+---------
   1 | Meier  | {mo,di,mi,do,fr} | 27:00
   2 | Müller | {fr}             | 01:30
```