# SQL-Vertiefung

## VU Datenbanksysteme

Reinhard Pichler

Arbeitsbereich Datenbanken und Artificial Intelligence
Institut für Informationssysteme
Technische Universität Wien

Wintersemester 2015/16

# Gliederung

# SQL-Programmteile in der Vorlesung

- Folien:
  - Enthalten viele Programmausschnitte
  - Programme manchmal nur auszugsweise auf den Folien wiedergegeben (immer nur die „wesentlichen“ Teile).
  - Durch das Auslassen von „unwesentlichen“ Details sind die Programme auf den Folien eventuell nicht ablauffähig.
- SQL-Sourcen:
  - Auf der DBS Homepage finden Sie vollständige Sourcen. Diese wurden unter PostgreSQL 9.1 getestet.

# Gliederung

# Constraint Typen

**CHECK** Zulässiger Wertebereich für eine Spalte oder eine Gruppe von Spalten (derselben Zeile!)

**NOT NULL** Spalte darf nicht den Wert **NULL** haben

**PRIMARY KEY** Eine Spalte oder eine Gruppe von Spalten wird als Primärschlüssel dieser Tabelle definiert

**UNIQUE** Die Werte in einer Spalte oder in einer Gruppe von Spalten müssen eindeutig sein.

**FOREIGN KEY** Eine Spalte oder eine Gruppe von Spalten wird als Fremdschlüssel auf eine andere Tabelle oder auch auf diese Tabelle definiert.

# Definition von Constraints

- Innerhalb/außerhalb der Spaltendefinition:
  Spalten- vs. Tabellen-Constraints
- Teil der Tabellen-Definition oder nachträglich:
  **CREATE TABLE** vs. **ALTER TABLE ADD CONSTRAINT**
- Name des Constraints:
  Benutzerdefiniert vs. vom System vergebener Name
- Änderungen:
  ENABLE, DISABLE (In Oracle auf verschiedene Constraints anwendbar; in PostgreSQL nur für Triggers möglich)
- Löschen von Constraints:
  **ALTER TABLE** ... **DROP CONSTRAINT** ...
  (Foreign Keys) **DROP TABLE** ... **CASCADE**

# Beispiel

## Beispiel 1

```
CREATE TABLE Professoren
(PersNr INTEGER PRIMARY KEY,
 Name    VARCHAR(30) NOT NULL,
 Rang    CHAR(2) CHECK (Rang in ('C2','C3','C4')),
 Raum    INTEGER UNIQUE );

CREATE TABLE Vorlesungen
(VorlNr       INTEGER PRIMARY KEY,
 Titel        VARCHAR(30),
 SWS          INTEGER,
 gelesenVon   INTEGER REFERENCES Professoren (PersNr) )
```

# Beispiel

## Beispiel 2

```
DROP TABLE Professoren CASCADE;
-- löscht Foreign Key in Tabelle Vorlesungen

CREATE TABLE Professoren
(PersNr INTEGER,
 Name   VARCHAR(30) CONSTRAINT nn NOT NULL,
 Rang   CHAR(2),
 Raum   INTEGER,
 CONSTRAINT prof_pk PRIMARY KEY (PersNr),
 CONSTRAINT rang_ch CHECK (Rang IN ... ),
 CONSTRAINT raum_un UNIQUE (Raum) );
```

# Beispiel

In Oracle:

## Beispiel 3

```
ALTER TABLE Vorlesungen
ADD CONSTRAINT prof_fk FOREIGN KEY (gelesenVon)
REFERENCES Professoren (PersNr) NOVALIDATE;
-- Bemerkung: Weil die Professoren Tabelle leer ist, ist dieser Foreign Key für
die Vorlesungen im Moment nicht gültig. Referential Integrity wird nur für neue
Vorlesungen geprüft.

INSERT INTO Vorlesungen VALUES ...; -- scheitert!
INSERT INTO Professoren VALUES ...;
ALTER TABLE Vorlesungen
ENABLE VALIDATE CONSTRAINT prof_fk;
-- überprüft Referential Integrity für alle Vorlesungen
```

# Deferred Constraints

Zyklische Foreign Key Beziehungen, z.B.:

- Zwei Tabellen: Abteilungen, Mitarbeiter
- Jeder Mitarbeiter ist einer Abteilung zugeordnet
- Jede Abteilung hat einen Chef (unter den Mitarbeitern)

- Drei Probleme
  1. **CREATE TABLE**: Wie FK der ersten Tabelle definieren?
  2. **INSERT**: Wie kann man z.B. neue Abteilungen einfügen?
  3. **DROP TABLE**: Wie löscht man FK auf dieselbe Tabelle?
- Lösung:
  - Constraints nachträglich einführen mit **ALTER TABLE**
  - *Deferred Constraints*: Überprüfung erst beim „commit“
  - **DROP CONSTRAINT** oder **CASCADE**

# Beispiel

## Beispiel 4

```
CREATE TABLE Abteilungen
( AbtNr INTEGER PRIMARY KEY,
  Chef INTEGER,
  ... );

CREATE TABLE Mitarbeiter
( PersNr INTEGER PRIMARY KEY,
  Name VARCHAR(30),
  AbtNr INTEGER CONSTRAINT abt_fk
    REFERENCES Abteilungen (AbtNr)
    DEFERRABLE INITIALLY DEFERRED);

ALTER TABLE Abteilungen ADD CONSTRAINT chef_fk
  FOREIGN KEY (Chef) REFERENCES Mitarbeiter (PersNr)
  DEFERRABLE INITIALLY DEFERRED;
```

# Beispiel

## Beispiel 5

```
INSERT INTO Abteilungen VALUES(1001, 102, ... );
INSERT INTO Abteilungen VALUES(1002, 203, ... );
INSERT INTO Abteilungen VALUES(1003, 301, ... );
INSERT INTO Mitarbeiter VALUES(101, ... , 1001);
INSERT INTO Mitarbeiter VALUES(102, ... , 1001);
-- etc.

COMMIT;

DROP TABLE Mitarbeiter CASCADE;
DROP TABLE Abteilungen CASCADE;
-- Beim Löschen der Abteilungen-Tabelle wäre CASCADE nicht nötig gewe-
sen, weil die Mitarbeiter-Tabelle zu diesem Zeitpunkt schon gelöscht ist).
```

# Gliederung

# Idee und Definition

- Problem
  - Manchmal benötigt man Integer PKs, deren Wert nicht wirklich relevant ist (er muss nur eindeutig sein).
  - Beispiele: PersonalNr, AusweisNr, MitgliederNr, etc.
- Lösung in SQL: **SEQUENCE**
  - Definition der Sequence mit Wertebereich, Start-Wert, Schritt-Größe, etc.
  - Funktion **nextval**: Sequenz erhöhen und anschließend auslesen.
  - Funktion **currval**: Aktuellen Wert auslesen.
- Definieren, Löschen einer Sequence
  **CREATE SEQUENCE** bzw. **DROP SEQUENCE**

# Beispiele

## Beispiel 6

```
DROP SEQUENCE pers_sequence;
CREATE SEQUENCE pers_sequence
  START WITH 3000
  INCREMENT BY 10
  MINVALUE 1
  MAXVALUE 10000
  NO CYCLE
  CACHE 1;

INSERT INTO Professoren VALUES(
  nextval('pers_sequence'), 'Prof A', 'C3', '4711');
INSERT INTO Professoren VALUES(
  nextval('pers_sequence'), 'Prof B', 'C3', '4712');
```

# Bemerkungen

- Default Werte: Wenn einer dieser Parameter nicht angegeben wird, wird der Default Wert genommen, z.B.: START WITH 1, INCREMENT BY 1, NO MINVALUE, NO MAXVALUE, NO CYCLE, ...
- Abwärts zählen: INCREMENT BY mit negativer Zahl
- CYCLE (vs. NO CYCLE): Wenn Grenze des Wertebereichs erreicht wird, setzt die SEQUENCE am anderen Ende des Wertebereichs fort, anstatt mit einem Fehler abzubrechen.
- CACHE $n$ ($n > 1$): SEQUENCE produziert $n$ Werte im Voraus (um sie dann im Cache zu speichern).

# Gliederung

# Single row vs. Aggregat-Funktionen

- Single row Funktionen
  - Character Funktionen
  - Numerische Funktionen
  - Konvertierungsfunktionen
  - Datumsfunktionen
- Aggregat-Funktionen
  - (wurden schon in der VU Datenmodellierung behandelt)
  - Berechnen einen Gesamtwert für mehrere Zeilen
  - z.B. **COUNT**, **MIN**, **MAX**, **SUM**, **AVG**

# Character Funktionen

- Concatenation: Aneinanderkettung von zwei Strings, z.B.:
  **SELECT** `'Postgre'||'SQL'`
- lower() und upper():
  Umwandlung in Klein- bzw. Großbuchstaben
- length():
  Liefert die Länge eines Strings.
- substring(string from for):
  **substring**('Thomas' **from** 2 **for** 3) – liefert 'hom'
- replace (string from to):
  **replace**('abcdefcdg', 'cd', 'XX') – liefert 'abXXefXXg'

# Numerische Funktionen

- „alles was ein Taschenrechner kann“, z.B.
  - sqrt(x), power(x,y)
  - exp(x), ln(x), log(b, x)
  - cos(x), sin(x), tan(x), atan(x), etc.
  - abs(x), sign(x)
- round():
  Runden (optional: auf bestimmte Anzahl von Dezimalstellen)
  - round(105.75) $--$ *liefert 106*
  - round(105.75, 1) $--$ *liefert 105.8*
  - round(105.75, $-1$) $--$ *liefert 110*

# Konvertierungsfunktionen

- Aufgabe: Umwandeln von Daten zwischen verschiedenen Datentypen
- z.B.: to_char(), to_number()
  - to_char(17) −− *liefert '17'*
  - to_char(12345.678, `'99,999.99'`) −−*liefert '12,345.68'*
  - to_number(`'17'`) −−*liefert 17*
  - to_number(`'-12,345.67'`,`'99,999.99'`): −12345.67
  - to_number(`'123.45'`,`'99.99'`): Error!
  - to_number(`'123.49'`,`'999.9'`): 123.4
- viele weitere Funktionen

# Gliederung

# Datentyp DATE

- Dient zum Speichern von Datum (Tag, Monat, Jahr)
- Keine Uhrzeit (SQL Standard)
- **CURRENT_DATE** liefert aktuelles Datum (laut SQL Standard: zu Beginn der Transaktion)
- to_char und to_date Funktionen bieten viele Formatierungsmöglichkeiten

# to_char() Funktion

- Beispiele:
  **SELECT** to_char(geboren) **FROM** kunden;
  **SELECT** to_char(abgeschickt) **FROM** bestellungen;
- Weitere Beispiele:
  to_char(Geboren, 'MONTH DD, YYYY')
  to_char(Geboren, 'DD-MON-YYYY')
  to_char(Geboren, 'Day, DD.MM.YY')
  to_char(CURRENT_DATE, 'DD-MON-YYYY')

# to_date() Funktion

- Beispiel:
  UPDATE KUNDEN SET Geboren =
  to_date('12-JUN-1976') WHERE KundenNr = 1001;
- Weitere Beispiele:
  to_date('12-JUN-1976')
  to_date('12.06.1976', 'DD.MM.YYYY')
  to_date('October 3, 1974', 'Month DD, YYYY')

# age() Funktion

Differenz zw. zwei Datumsangaben in Jahren/Monaten/Tagen

## Beispiel 7

```
select age(current_date,
     to_date('01 Sep 2012', 'DD Mon YYYY'));
-> "2 years 1 mon 16 days"
   (eingegeben am 17.10.2014)

select age(to_date('01 Nov 2014', 'DD Mon YYYY'),
     to_date('05 Dec 2010', 'DD Mon YYYY'));
-> "3 years 10 mons 27 days"

select age(to_date('01 Nov 2012', 'DD Mon YYYY'),
     to_date('05 Dec 2013', 'DD Mon YYYY'));
-> "-1 years -1 mons -4 days"
```

# Datentypen TIME und TIMESTAMP

- TIME: zum Speichern der Zeit
- TIMESTAMP: zum Speichern von Datum und Zeit
- **CURRENT_TIME** bzw. **CURRENT_TIMESTAMP**: liefert aktuelle Zeit bzw. Zeit+Datum (laut SQL Standard: zu Beginn der Transaktion).
- to_char bzw. to_timestamp Funktionen bieten viele Formatierungsmöglichkeiten.

# extract() Funktion

## Beispiel 8

```
SELECT EXTRACT (YEAR FROM Datum) AS Jahr,
  EXTRACT (MONTH FROM Datum) AS Monat,
  EXTRACT (DAY FROM Datum) AS Tag
  FROM Bestellungen WHERE KundenNr = 1003;
```

# Gliederung

# Idee

- Komplexe Anfragen:
  - (mehrfach) geschachtelte SELECTs
  - Zerlegung der Anfrage in “überschaubare” Teile erhöht die Lesbarkeit.
- Hierarchische Beziehungen in einer DB:
  - Attribut „Chef“ in der Mitarbeiter-Tabelle
  - Uni-DB: Vorlesung als Voraussetzung einer anderen Vorlesung.
  - Attribut „Bestandteil-von“ in Bauteile-Tabelle
  - E-Mail Threads
- Anfragen, die die gesamte Hierarchie durchlaufen:
  - in SQL-92 nicht vorhanden (Rekursion fehlt)
  - erst ab SQL-99
  - Oracle: CONNECT BY

# WITH Queries

## WITH Ausdrücke

- auch ”Common Table Expressions (CTE)” genannt:
  sind temporäre Resultate/Ausdrücke
  (im Scope der jeweiligen Ausführung sichtbar)
- ohne ”RECURSIVE”: dient nur der besseren Lesbarkeit
- mit ”RECURSIVE”: rekursive Abfragen möglich

# Beispiel

## Beispiel 9

```
WITH regional_sales AS (
  SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
  ), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales >
      (SELECT SUM(total_sales)/10 FROM regional_sales)
  )
SELECT region,
  product,
  SUM(quantity) AS product_units,
  SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

# WITH RECURSIVE Queries

1. CTE Ausdruck in *anchor* und *recursive* teilen;
2. *Anchor* generiert initiales result set ($T_0$);
3. Rekursiver Teil mit $T_i$ als Input liefert als Output $T_{i+1}$;
4. Schritt 3 wiederholen, bis keine neuen Ergebnisse mehr dazukommen;
5. Gesamtes result set entspricht UNION bzw. UNION ALL von $T_0$ bis $T_n$.

# Beispiel

## Beispiel 10

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity
    FROM parts
    WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

Vorsicht: Endlosschleife, falls der rekursive Teil immer (neue) Tupel liefert (weniger gefährlich bei UNION als UNION ALL)!

# Beispiel

## Beispiel 11

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity
    FROM parts
    WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

Vorsicht: Endlosschleife, falls der rekursive Teil immer (neue) Tupel liefert (weniger gefährlich bei UNION als UNION ALL)!