Praktikum Informationssysteme

Anmeldedaten Gowron:

- Nachname auf 8 Stellen
- Passwort: Matrikelnummer

Kommandos:

- psql: ruft SQL Interpreter auf
- \q: beendet den SQL Interpreter
- \h: listet alle SQL Befehle auf
- \c [nutername]: zu Benutzer wechseln
- \z: Tabellen eines Nutzers anzeigen

Derjenige, der die Datenbank angelegt hat, hat das Recht, Rechte zu vergeben. Aber es können natürlich auch die Rechte auf die Datenbank vergeben werden. Kann ebenfalls auf Views angewendet werden.

```
GRANT SELECT ON WS14 TO public;
```

```
Sichten sind wichtig, da die Daten immer aktuell sind, da sie keine echten Tabellen sind. So können andere Nutzer nur auf Teildaten einer Tabelle einfach zugreifen.
```

```
GRANT SELECT ON Dienstags TO altaner;
```

Regelsystem

Vorbereitungen:

```
CREATE TABLE Gegenstand(ID INT, Bezeichnung VARCHAR(30));
CREATE TABLE LogAnz(Anz INT);
INSERT INTO LogAnz VALUES(0);
```

```
Jedes mal soll bei insert automatisch eine Anzahl in LogAnz gelogged werden.
```

```
CREATE rule protokoll
AS ON INSERT TO Gegenstand
do also UPDATE LogAnz SET Anz = Anz+1;
```

```
also kann weggelassen werden, da also Standard ist. \\ Test:
```

```
 INSERT INTO Gegenstand VALUES(1, 'Computer');
INSERT INTO Gegenstand VALUES(2, 'Drucker');
SELECT * FROM LogAnz;
```

Nun soll als Regel die ID aus Gegenstand, die bei insert erstellt wird auch in LogAnz erstellt werden. Dazu brauchen wir virtuelle Tabellen (new, old). New enthält die Werte und Spalten, die neu beim Abarbeiten der Regel hinzugekommen sind. Old sammelt die Werte, die von insert und delete betroffen waren. Bei update lassen sich beide Regeln benutzen. Zuerst wird die zuvor erstellte Regel entfernt und eine neue Tabelle Log erstellt:

```
DROP rule protokoll ON Gegenstand;
CREATE TABLE Log(ID INT);
```

```
CREATE rule protokoll
AS ON INSERT TO Gegenstand
do also INSERT INTO Log VALUES(NEW.ID);
```

Man kann bei Sichten kein insert anwenden, da es ja nur die (Teil-)Darstellung einer Tabelle ist. Möglich ist es, indem man via Regel in die eigentliche Tabelle einfügt:

```
CREATE rule insertRegel AS
ON INSERT TO Dienstags do instead
INSERT INTO WS14 VALUES(NEW.Wochentag, NEW.Beginn, NEW.Ende, NEW.Nr,
NEW.Name, NEW.Dozent);
```

Delete Regel lautet (bei der Where Bedingungen bietet sich immer der Primary Key an. Eigentlich müssten via and Bedingung alle Spalten definiert werden (old.Wochentag and …):

```
CREATE rule deleteRegel AS
ON DELETE TO Dienstags do instead
DELETE FROM WS14 WHERE Nr = OLD.Nr;
```

Update:

```
CREATE rule updateRegel AS
ON UPDATE TO Dienstags do instead
UPDATE WS14 SET Wochentag=NEW.Wochentag,
Beginn=NEW.Beginn,
Ende=NEW.Ende,
Nr=NEW.Nr,
Name=NEW.Name,
Dozent=NEW.Dozent
WHERE Nr=OLD.Nr;
```

Test:

```
UPDATE Dienstags SET Beginn = 11 WHERE Dozent = 'Siegmund';
SELECT * FROM Dienstags;
```

```
DELETE FROM Dienstags WHERE Name='Foo';
```

```
Was macht folgende Regel:
```

```
CREATE rule ReK AS
ON INSERT TO Doppelt do
INSERT INTO Doppelt VALUES(NEW.A+1, NEW.B);
```

```
Bei jedem Einfügen wird eine weitere Zeile einfügen und so weiter… D.h.
diese Regel ist rekursiv. Mit do instead wird die Tabelle nicht überlaufen,
da nur der zweite Teil der Regel ausgeführt wird. Instead wird das Problem
allerdings vertagen, da gewartet wird.
```

Schlüssel

Wenn zwei Tabellen mittels primary und foreign key miteinander verbunden sind, kann es beim Update von Werten des Schlüssel beider Tabellen zu Problemen führen, auch wenn die Bedingung zwischen primary und foreign key beim Update inhaltlich nicht verletzt wird.
<note tip>constraint dient dazu einen Namen vergeben zu können. So können z.B. Schlüssel ganz einfach wieder gelöscht werden.</note> Erstellen einer Tabelle und Verknüpfung mittels keys - Beispiel:

```
ALTER TABLE WS14 ADD PRIMARY KEY(Nr);

CREATE TABLE HS13(Wochentag CHAR(2), Uhrzeit INT, Vorlesung VARCHAR(6),
PRIMARY KEY(Wochentag, Uhrzeit),
CONSTRAINT Fremdschluessel
FOREIGN KEY(Vorlesung) REFERENCES WS14);

INSERT INTO HS13 VALUES('Di', 10, '5102V');
INSERT INTO HS13 VALUES('Di', 11, '5102V');
```

```
Update beider Tabellen führt zu Problem, da Integritätsprüfung erst nach
kurz vor dem commit ausgeführt werden soll:
```

```
BEGIN TRANSACTION;
UPDATE WS14 SET Nr = '5103V' WHERE Nr = '5102V';
UPDATE HS13 SET Vorlesung = '5103V' WHERE Vorlesung = '5102V';
commit;
```

```
Daher muss bei der Definition des foreign key die Bedingung deferrable
initially deferred gesetzt werden:
```

```
ALTER TABLE HS13 DROP CONSTRAINT Fremdschluessel;
ALTER TABLE HS13 ADD CONSTRAINT Fremdschluessel
FOREIGN KEY(Vorlesung) REFERENCES WS14 deferrable initially deferred;
```

```
Zweite Variante: Definieren der Fremschlüsselbedingung als verzögerbar, aber
```

```
sofort standardmäßig als sofort-ausführend, um dannbeim Ausführen einer Transaktion zu definieren, dass die Bedingung verzögert werden soll.
```

```
ALTER TABLE HS13 ADD CONSTRAINT Fremdschluessel
FOREIGN KEY(Vorlesung) REFERENCES WS14 deferrable initially immediate;
BEGIN TRANSACTION;
SET constraints Fremdschluessel deferred;
UPDATE WS14 SET Nr = '5102V' WHERE Nr = '5103V';
UPDATE HS13 SET Vorlesung = '5102V' WHERE Vorlesung = '5103V';
commit;
```

<note tip>deferrable definiert, dass eine Aktion verzögert ausgeführt werden darf. initially meint, was beim Start der Aktion ausgeführt werden soll. Dazu dienen die Parameter immediate (sofort) und deferred (verzögert).</note>

Transaktionen

Zwei Insert-Aktionen werden parallel mit Transaktionen gestartet.
Beispiel 1. Fenster:

```
BEGIN;
INSERT INTO WS14 VALUES('Fr',10,12,'5555V','Java','Mueller');
commit;
```

```
Beispiel 2. Fenster (parallel):
```

```
BEGIN;
INSERT INTO WS14 VALUES('Fr',10,12,'5555V','SQL','Mueller');
```

```
Mit jeweils select * from WS14 sehen wir bereits die eingefügten Werte. Was passiert nun, sobald commit; eingegeben wird? Per Voreinstellung wissen andere Transaktionen, was gerade passiert. \\ Es werden zwei Transaktionen geöffnet und jeweils werden bei einem Insert die gleichen Primary Keys eingegeben. Ein Insert funktioniert, beim zweiten wird das Prozedere angehalten, da die eine Transaktion wartet, ob die andere committed oder aborted. Je nach dem, was nun passiert, meldet die andere Transaktion einen Fehler oder einen Erfolg.
```

Isolationsgrade

Voreinstellung für Transaktionen ist read committed . <note tip>Isolationsgrade bestimmen, was andere Transaktionen von dem was um sie herum passiert, sehen können.</note>

- read committed: andere Transaktionen können Lesen, was passiert.
- serializable: Abschotten der Transaktionen von der Außenwelt, so als wären sie hintereinander ausgeführt worden, aber lost update problem.

```
BEGIN;
SET TRANSACTION isolation level serializable;
INSERT INTO WS14 VALUES('Di', 16,18,'5500V','Datenbanken','Schulze');
commit;
```

```
Obwohl eine Transaktion mit commit; abgeschlossen wird, sieht die andere Transaktion bei serializable das nicht. Was passiert nun bei einem update-Befehl in zwei Transaktionen, die das gleiche updaten wollen? Mit serializable wartet die andere Transaktion, was die erste macht, d.h. hier kommt auch ein Error, wenn die erste Transaktion committed. \\ Bei read committed kommt kein Fehler, sondern es wird ein update ausgeführt, aber die erste Transaktion hat das update erfolgreich ausgeführt (lost update Problem).
```

Sperren

- Es gibt 8 verschiedene Sperren: Tabellen- und Zeilensperren
- Sperren bleiben bis zum Ende der Transaktion erhalten und können nicht entsperrt werden (2-Phasen-Sperr-Protokoll)
- Sperren sind im Grunde nur Namen, die auf eine Tabelle geklebt wird. Es wird also nur definiert, welche Sperre mit welcher in Konflikt steht
- Bestimmte SQL-Befehle richten automatisch bestimmte Sperren ein, aber Tabellensperren können auch individuell festgelegt werden

Übersicht möglicher Sperren

Konflikte von PostgreSQL-Sperren untereinander	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Automatische Sperren

- ACCESS SHARE: SELECT
- ROW SHARE: SELECT FOR UPDATE
- ROW EXCLUSIVE: UPDATE, DELETE, INSERT
- SHARE UPDATE EXCLUSIVE: VACUUM
- SHARE: CREATE INDEX
- SHARE ROW EXCLUSIVE: -
- EXCLUSIVE: -
- ACCESS EXCLUSIVE: ALTER TABLE, DROP TABLE

Anwendung

Beispiel für Sperren anfordern:

```
LOCK WS14 IN ROW exclusive mode;
/*Struktur: lock [tabellenname] in [modusname] mode */
```

- sukzessiv können mehrere Sperren angefordert werden
- Wenn durch Einsatz von Sperren bei Transaktionen in Konflikt stehen, wartet die andere Transaktion
- Sperren dienen dazu, Tabellen bei vielen Operationen zu sperren
- Sperren innerhalb derselben Transaktion stehen nie in Konflikt
- Wenn sich Sperren gegenseitig blockieren, wird eine Transaktion beendet (dead lock)

Beispiel für einen Dead Lock

Transaktion 1

```
BEGIN;
SELECT * FROM WS14;
ALTER TABLE WS14 ADD Zusatz INT;
/* wartet... */
```

```
Transaktion 2
```

```
BEGIN;
SELECT * FROM WS14;
ALTER TABLE WS14 ADD bla INT;
/* abgebrochen wegen Deadlock */
```

⇒ Dead lock kann durch starke Sperren verhindert werden.

Vererbung

Wiederholung: 3 Tabellen erstellen

```
CREATE TABLE Uni_Bedienstet(PersNr INT, Vorname VARCHAR(30), Name
VARCHAR(30), ZiNr INT, TelNr INT);
CREATE TABLE Professor(Lehrgebiet VARCHAR(30), STATUS CHAR(2))
inherits(Uni_Bedienstet);
/* inherits sagt, dass alle Attribute von Professor an Uni_Bedienstet
vererbt werden */
CREATE TABLE Assistent(Projekt VARCHAR(30), Vertragsende DATE)
inherits(Uni_Bedienstet);
INSERT INTO Uni_Bedienstet VALUES (12345, 'Klaus', 'Maier', 210, 2016);
INSERT INTO Professor VALUES(12346, 'Helga', 'Huber' 314, 2020, 'Politik',
'W2');
INSERT INTO Assistent VALUES(12347, 'Uwe', 'Schulze',221,2160, 'SQL-
Projekt', '31.03.2015');
```

<note tip>Reihenfolge bei Insert in table mit Vererbung: erst die Generalisierung, dann die eigenen Attribute.</note>

Select mit vererbten Tabellen

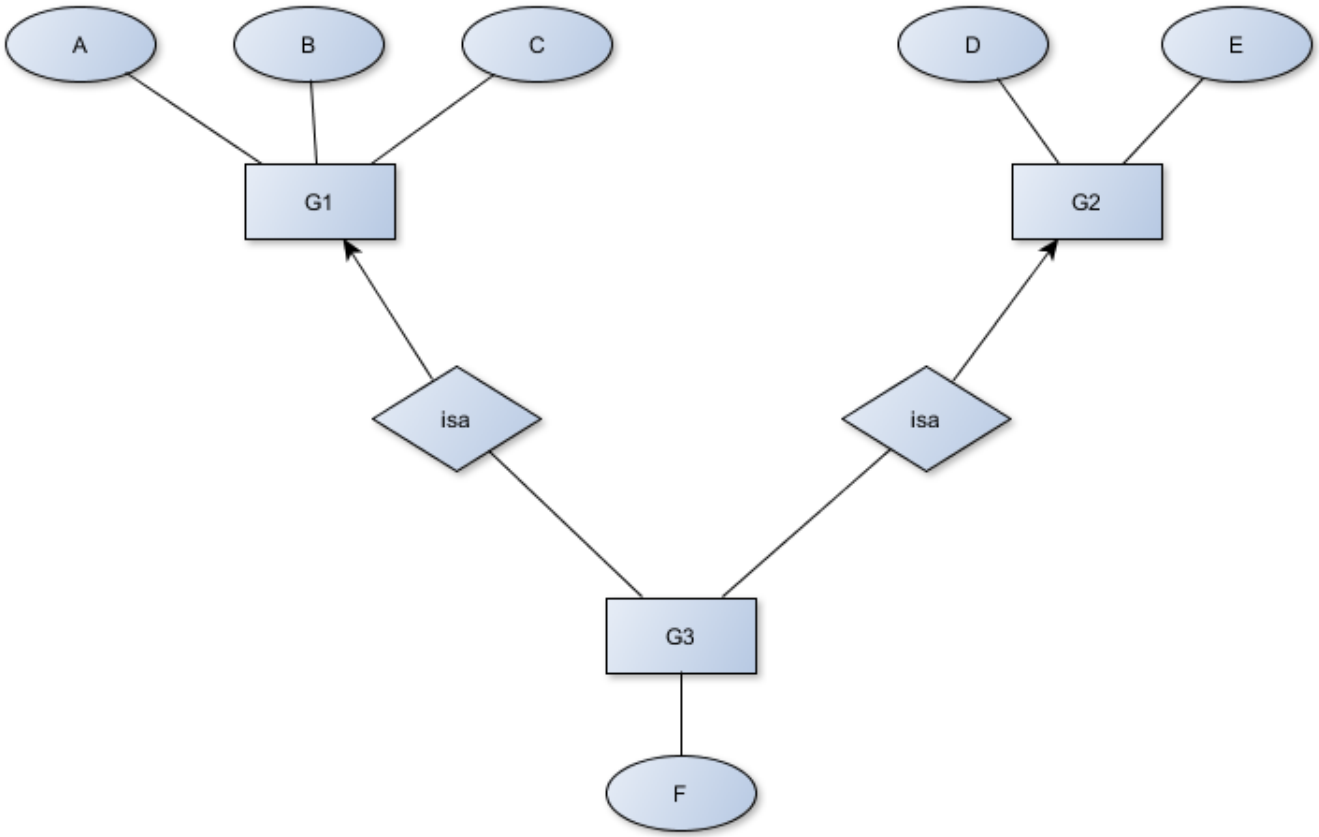
Mit select * from Uni_Bedienstet sind alle Personen aufgelistet, da diese ja vererbt werden. Bei den jeweiligen anderen Tabellen sind nur die einzelnen Personeneinträge gelistet. D.h. bei insert werden die Spezialisierungen in der Generalisierung angezeigt, auch wenn diese nicht Teil davon sind.
Eine Variante des select Befehls: only. Dann werden keine Vererbungen bei select ausgegeben, sondern nur Einträge in dieser Tabelle.

```
SELECT * FROM ONLY Uni_Bedienstet;
```

Update / Delete mit vererbten Tabellen

Wenn aus vererbten Tabellen Einträge gelöscht werden, sind diese auch in der generalisierten Tabelle gelöscht.
Sollte ein Wert mit Update aktualisiert werden, wird dieser auch in der Generalisierung sichtbar.

Beispiel: eine Spezialisierung und 2 Generalisierungen



Mögliche Probleme: Wenn 2 Datentypen gleich sind bei der Generalisierung, denn in der

Spezialisierung werden nur Werte angezeigt, während sie ja lediglich in den Generalisierungen fest geschrieben werden. Dort besteht ja dann kein Problem.
Mehrfache Vererbung ist möglich. Es geht so lange, wie alle gemeinsamen Datentypen in den Generalisierungen gleiche Datentypen haben. Sollte diese allerdings unterschiedlich sein, gibt es ein Problem.

Zusatzspalte: Woher kommen die Werte

OIDs: Object Identifier. Identifieren jedes Objekt in der Datenbank und vergeben eine eindeutige Nummer. Beispiel für eine OID: INSERT 39654 [1(=Anzahl der Zeilen)].

```
SELECT oid, * FROM Uni_Bedienstet;
```

```
Table-OID: Eindeutige Nummer für Tabellen.
```

```
SELECT oid, tableoid, * FROM Uni_Bedienstet;
```

```
Aber Zahlen sind wirr, d.h. es gibt auch Systemtabellen, die die Namen der
jeweiligen Tabellen ausgeben.
```

```
SELECT relname AS Tabelle, Uni_Bedienstet, * FROM Uni_Bedienstet, pg_class
WHERE Uni_Bedienstet.tableoid = pg_class.oid;
```

Funktionen SQL

Beispiele:

```
SELECT 3+4:
/* ergibt eine Spalte mit Ergebnis 7*/
SELECT SQRT(2);
```

- Funkionen: Haben ein Argument, eine Zahl und ein Ergebnis.
- Aggregatfunktionen: Nehmen mehrere Werte und verdichten diese zu einem Wert, bsp. die Summer einer Spalte.

Aufbau einer Funktion

Eine Artikeltabelle soll bei Verkaufspreis 19% Mwst. erhalten.
Möglichkeit 1:

```
SELECT Bezeichnung, Verkaufspreis AS Netto,
  Verkaufspreis * 0.19 AS Mwst,
  Verkaufspreis + Verkaufspreis * 0.19 AS Brutto FROM Artikel;
```

```
Möglichkeit 2:
```

```
CREATE FUNCTION mwst(NUMERIC) RETURNS NUMERIC AS $$
    SELECT round($1 * 0.19, 2);
$$ LANGUAGE SQL;
```

- Die eingesetzten Datentypen müssen nicht gleich sein, d.h. es kann ein anderer Datentyp returned werden, als ursprünglich eingegeben wird.
- Dollar 1: Platzhalter für das erste Argument, das übergeben wird.
- language muss definiert werden, da mehrere Sprachen möglich sind.
- Dollar Dollar: eine Art von Anführungszeichen, die die Funktion definieren.

```
SELECT
    Artikelnr,
    Bezeichnung,
    Verkaufspreis AS Netto,
    mwst(Verkaufspreis) AS MwSt,
    Verkaufspreis + mwst(Verkaufspreis) AS Brutto
FROM Artikel;
```

```
Erweiterung round:
```

```
SELECT ... AS Netto, round(mwst(Verkaufspreis),2) AS mwst, round(...,2) AS
Brutto FROM Artikel;
```

SQL Procedural Language

Aufgabe 1: String mit jeweils einem Leerzeichen zwischen den Buchstaben ausgeben.

```
CREATE FUNCTION spaceout(VARCHAR) RETURNS VARCHAR AS $$
DECLARE
    str VARCHAR;
    ret VARCHAR;
    len INT;
BEGIN
    str:= UPPER($1); /* Alles in Großbuchstaben */
    ret:= '';
    len:= LENGTH(str);
    FOR i IN 1..len loop
       ret:= ret || substr(str, i, 1) || ' ';
    END loop;
    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

```
Aufgabe 2: Eine Funktion soll 2 String auf Ähnlichkeit überprüfen und die
unterschiedlichen Stellen sollen als Anzahl ausgegeben werden. ⇒ akin
```

- abs: absolute Zahl
- <>: ungleich

```
CREATE FUNCTION diff(VARCHAR, VARCHAR) RETURNS INT AS $$
DECLARE
    len1 INT;
    len2 INT;
    minLen INT;
    diffLen INT;
    diffCount INT;
BEGIN
    len1:= LENGTH($1);
    len2:= LENGTH($2);
    IF len1 < len2 THEN
        minLen:= len1;
    ELSE
        minLen:= len2;
    END IF;
    diffLen:= abs(len1 - len2);
    diffCount:= 0;

    FOR i IN 1..minLen loop
       IF substr($1, i, 1) <> substr($2, i, 1) THEN
           diffCount:= diffCount + 1;
       END IF;
    END loop;
    diffCount:= diffCount + diffLen;
    RETURN diffCount;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION akin(VARCHAR, VARCHAR) RETURNS BOOLEAN AS $$
    SELECT diff($1, $2) <= 1;
$$ LANGUAGE SQL;
```

Operatoren definieren

```
CREATE operator ~= (
    PROCEDURE = akin,
    leftarg = VARCHAR,
    rightarg = VARCHAR);
```

```
Möglich sind dann Abfragen der Art select * from Artikel where Hersteller ~=
'Maierhofer';
```

Trigger

Sind Ereignisse, die etwas auslösen.
Erst muss eine Funktion erstellt werden.

```
CREATE FUNCTION capitalize() RETURNS TRIGGER AS $$
/* keine Parameter müssen eingegeben werden */
BEGIN
  NEW.bezeichnung = initcap(NEW.bezeichnung);
  /* Auf new Tabellen können Änderungen vorgenommen werden */
  RETURN NEW;
END $$ LANGUAGE plpgsql;
```

```
Daraus wird nun der Trigger erstellt:
```

```
CREATE TRIGGER trigger_cap
BEFORE INSERT OR UPDATE
ON aritkel
FOR each ROW
/*standardmäßig*/
EXECUTE PROCEDURE capitalize();
```

Aufgabe 1

Tabelle mit Terminen erstellen und bei insert oder update sollen die Wochentage auch in englisch verstanden werden, aber automatisch auf deutsche Abkürzungen übertragen werden.

```
CREATE TABLE Termine(Wochentag VARCHAR(3), Uhrzeit INT, Notiz VARCHAR(100));
INSERT INTO Termine VALUES
('Mo', 10, 'Besprechung'),
('Di', 8, 'Wichtig: Abgabeschluss Report'),
('Di', 14, 'Mit dem Chef wichtige Details zum Projekt bereden'),
('Di', 15, 'Maier kontaktieren'),
('Fr', 17, 'Dringend: Geschenke besorgen');
```

```
Erstellen der Funktion und des Triggers:
```

```
CREATE FUNCTION tage_en2de() RETURNS TRIGGER AS $$
DECLARE
    tag VARCHAR;
BEGIN
    NEW.wochentag = initcap(NEW.wochentag);
    tag = NEW.wochentag;
    IF tag ='Mon' THEN
        tag = 'Mo';
    elsif tag = 'Tue' THEN
        tag = 'Di';
    elsif tag = 'Wed' THEN
        tag = 'Mi';
    elsif tag = 'Thu' THEN
        tag = 'Do';
    elsif tag = 'Fri' THEN
```

```
        tag = 'Fr';
    elsif tag = 'Sat' THEN
        tag = 'Sa';
    elsif tag = 'Sun' THEN
        tag = 'So';
    elsif (tag = 'Mo') OR (tag = 'Di') OR (tag = 'Mi') OR
          (tag = 'Do') OR (tag = 'Fr') OR (tag = 'Sa') OR (tag = 'So') THEN
        -- nichts zu tun
    ELSE
        raise exception 'Fehlerhafter Wochentag';
    END IF;
    NEW.wochentag = tag;
    RETURN NEW;
END $$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_wochentage
BEFORE INSERT OR UPDATE
ON termine
FOR each ROW
EXECUTE PROCEDURE tage_en2de();
```

Aufgabe 2

Sorgen Sie jetzt noch dafür, dass Termine, in deren Notiz die Zeichenfolge „wichtig“ vorkommt, nicht mehr gelöscht werden können!

```
CREATE FUNCTION wichtig_schuetzen() RETURNS TRIGGER AS $$
BEGIN
    IF OLD.notiz ~* 'wichtig' THEN
    /* ~* bedeutet, dass wichtig in der Zeichenkette vorkommen muss */
        raise exception 'Kann wichtige Termine nicht loeschen!';
    END IF;
    RETURN OLD;
END $$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_wichtig
BEFORE DELETE
ON termine
FOR each ROW
EXECUTE PROCEDURE wichtig_schuetzen();
```

```
Test mit Löschen eines 14 Uhr Termins, wobei 2 14 Uhr Termine existieren,
wovon nur einer mit wichtig markiert ist: Kein Termin wird gelöscht, da ein
Befehl in SQL ganz oder gar nicht ausgeführt wird.
```