Informationssysteme

Sommersemester 2016

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Integrität / Trigger

Integritätskontrolle: Motivation

Idee/Beobachtung

- Abbildung der "Miniwelt" in relationales Modell dreht sich nicht nur um Speichern von Daten in Tabellen
- Wichtiger Aspekt ist die Kontrolle/Überwachung der Daten in den Relationen (und auch relationenübergreifend) zur Identifikation illegaler Zustände.

Realisierung

- Integritätsbedingungen (Constraints) spezifizieren akzeptable DB-Zustände
- Änderungen an der Datenbank werden zurückgewiesen, wenn sie entsprechend der Integritätsbedingungen als falsch bzw. ungültig erkannt werden.

Integritätskontrolle durch die Datenbank

DBS-basierte Integritätskontrolle

- größere Sicherheit
- vereinfachte Anwendungserstellung
- Unterstützung von interaktiven sowie programmierten DB-Änderungen
- leichtere Änderbarkeit von Integritätsbedingungen
- ggf. Leistungsvorteile

Arten von Integritätsbedingungen

Integritätsbedingungen abhängig vom Relationenmodell

- Primärschlüsseleigenschaft
- Referentielle Integrität für Fremdschlüssel
- Definitionsbereiche (Domains) für Attribute

Reichweite der Bedingung

- Attributwert-Bedingungen (z.B. Geburtsjahr $>$ 1900)
- Satzbedingungen (z.B. Geburtsdatum $<$ Einstellungsdatum)
- Satztyp-Bedingungen (z.B. Eindeutigkeit von Attributwerten)
- Satztypübergreifende Bedingungen (z.B. referentielle Integrität zwischen verschiedenen Tabellen)

Klar, je geringer die Reichweite, desto einfacher lassen sich Bedingungen überprüfen.

Arten von Integritätsbedingungen (2)

Statische vs. dynamische Bedingungen

- Statische Bedingungen (Zustandsbedingungen): beschränken zulässige DB-Zustände (z.B. Gehalt < 500000)
- Dynamische Integritätsbedingungen (Übergangsbedingungen): zulässige Zustandsübergänge (z.B. Gehalt darf nicht kleiner werden)
- Variante dynamischer Integritätsbedingungen: temporale IBs für längerfristig Bedingungen

Zeitpunkt der Überprüfbarkeit: unverzögert vs. verzögert

- Verzögerte Bedingungen lassen sich nur durch eine Folge von Änderungen erfüllen (typisch: mehrere Sätze, mehrere Tabellen) und
- Benötigen Transaktionsschutz (als zusammengehörige Änderungssequenzen)

Integritätsbedingungen

- =zusätzliches Sicherheitssystem
- Ziel: Dateninkonsistenzen vermeiden
- Strategie: versuche zu verhindern, dass inkonsistente Daten in die DB eingefügt werden
- Bedingungen an die möglichen Ausprögungen der Datenbank

Was wir bereits in der VL behandelt haben:

- keine zwei Tupel dürfen denselben Wert bezüglich ihres Schlüssels haben
- Kardinalitäten/Stelligkeiten von Beziehungen
- Generalisierung: jede Entität aus dem Untertyp muss auch im Obertyp enthalten sein
- Domänen (d.h. zulässige Werte) von Attributen
- Bedingungen an Zustandsübergänge der Datenbank

Einfache statische Integritätsbedingungen

nicht NULL

... persnr integer **NOT NULL** ...

Einschränkungen des Wertebereichs

... **check** Semester **between** 1 **and** 20

Aufzählungstypen

... **check** Rang **in** (‘C2‘,‘C3‘,‘C4‘) ...

Referentielle Integrität

Fremdschlüssel

- verweisen auf Tupel einer Relation
- z.B. gelesenVon in **Vorlesungen** verweist auf Tupel in **Professoren**

Referentielle Integrität bedeutet:

- Fremdschlüssel müssen auf existierende Tupel verweisen **oder** einen Nullwert enthalten.

Gegenbeispiel:

insert into Vorlesungen
values (5100, 'Agentenwesen', 4, 007);

Wie kann dieses "insert" verhindert werden?

Festlegung von Schlüsseln

unique:

- für alle Kandidatenschlüssel
- minimale Teilmenge von Attributen, die Tupel eindeutig definiert
- mehrere Kandidatenschlüssel für eine Relation sind möglich
- darf NULL sein

primary key:

- Primärschlüssel
- ein Schlüssel aus der Menge der Kandidatenschlüssel
- nur ein Primärschlüssel pro Relation
- impliziert not NULL

foreign key:

- Fremdschlüssel
- impliziert, dass NULL möglich ist, kann aber als not NULL definiert werden **unique foreign key**: modelliert 1:1-Beziehung

Referentielle Integrität in SQL

- Kandidatenschlüssel: **unique**
- Primärschlüssel: **primary key**
- Fremdschlüssel: **[foreign key] references**

Beispiel:

create table R
(α **integer primary key**,
...);

create table S
(...,
κ **integer references** R(α),
...);

Zusammengesetzte Schlüssel

create table R
(α,
β,
...,
primary key(α,β),
...);

create table S
(...,
κ_1,
κ_2,
...,
foreign key(κ_1,κ_2) references R(α,β)
...);

Kandidatenschlüssel

create table R
(α,
β,
γ,
...,
unique(β,γ)
...);

Beispiel: Schlüssel in Postgres

```
CREATE TABLE assistenten
(
    persnr integer NOT NULL,
    "name" character varying(30) NOT NULL,
    fachgebiet character varying(30),
    boss integer,
    CONSTRAINT assistenten_pkey PRIMARY KEY (persnr),
    CONSTRAINT assistenten_boss_fkey FOREIGN KEY (boss)
        REFERENCES professoren (persnr) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

MATCH SIMPLE erlaubt NULL-Werte in einem Teil des Schlüssels
MATCH FULL verhindert dies

Verhalten bei Änderungsoperationen

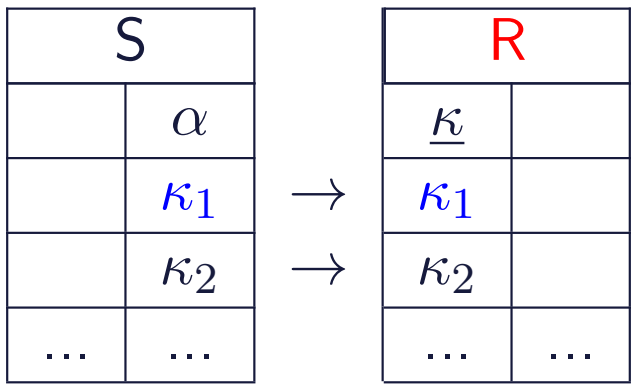
Bei Änderung von referenzierten Daten haben wir mindestens 3 Optionen:

1. **Zurückweisen** der Änderungsoperation (=Default)
2. **Propagieren** der Änderungen: **cascade**
3. Verweise auf <**unbekannt**> setzen: **set null**

Desweiteren in Postgres:

4. auf Defaultwert setzen: **set default**

Beispiel: Änderungsoperation auf R



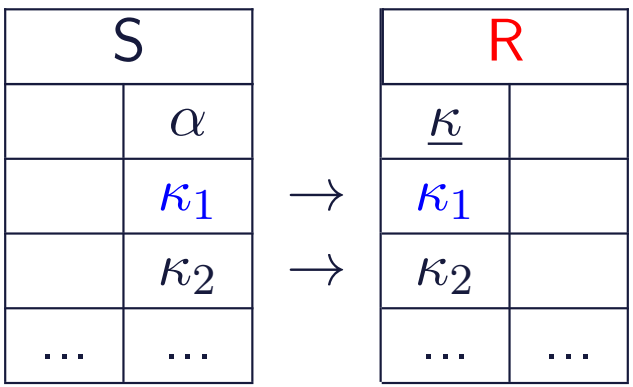
update R
set $\kappa = \kappa_1'$
where $\kappa = \kappa_1$

delete from R
where $\kappa = \kappa_1$

Wie soll damit umgegangen werden?

Option 1: Zurückweisen der Änderung

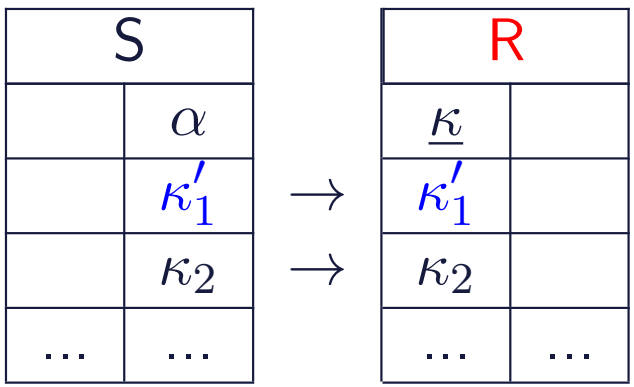
create table S
(...,
α **integer references** R(κ));



Ergebnis der Änderungsoperation: DB bleibt unverändert

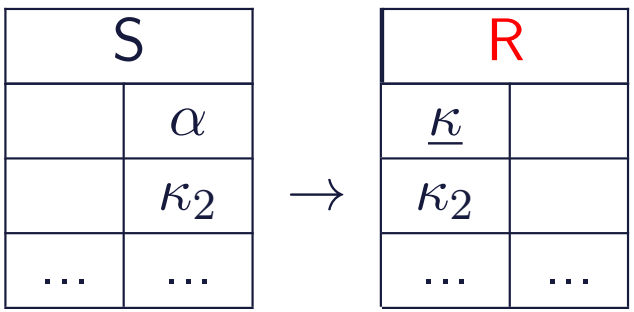
Option 2: Kaskadieren der Änderung

create table S
(....,
α **integer references** R(κ)
on update cascade);



Ergebnis der Änderungsoperationen:
Schlüssel in R **und** S geändert.

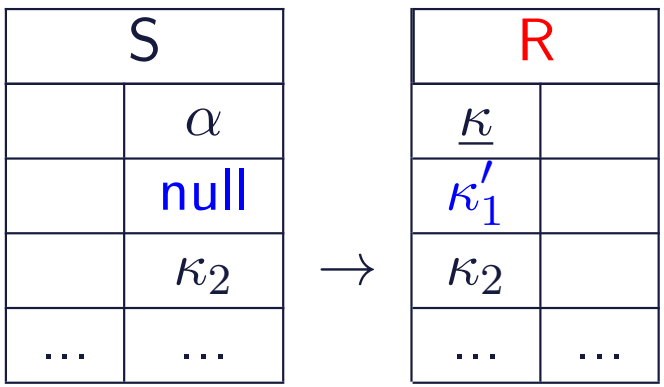
create table S
(....,
α **integer references** R(κ)
on delete cascade);



Ergebnis der Änderungsoperationen:
Schlüssel in R **und** S gelöscht.

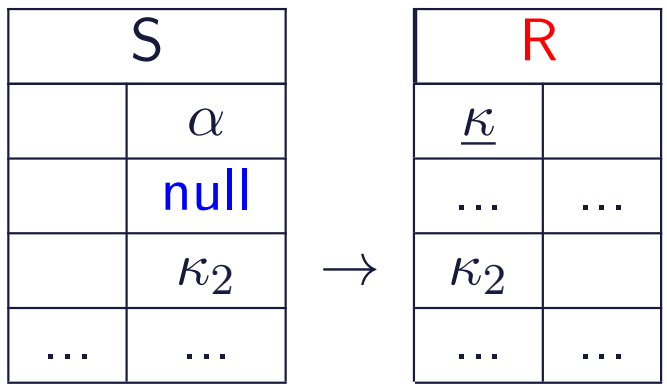
Option 3: Ändern und auf NULL setzen

create table S
(....,
α **integer references** R(κ)
on update set null);



Ergebnis der Änderungsoperationen: Tupel aus R auf κ_1', in S auf null geändert.

create table S
(....,
α **integer references** R(κ)
on delete set null);



Ergebnis der Änderungsoperationen: Tupel aus R gelöscht, Schlüssel in S auf null geändert.

Kaskadierendes Löschen

```
delete from Professoren where Name = 'Sokrates';

create table Vorlesungen
    (....,
     gelesenVon integer
                references Professoren(PersNr)
                on delete cascade);

create table hoeren
    (....,
     VorlNr integer
            references Vorlesungen(VorlNr)
            on delete cascade);
```

Das Univeritätsschema mit Integritätsbedingungen

```
create table Studenten
    (MatrNr    integer primary key,
    Name       varchar(30) not null,
    Semester   integer check Semester between 1 and 13);

create table Professoren
    ( PersNr   integer primary key,
    Name       varchar(30) not null,
    Rang       character(2) check (Rang in ('C2', 'C3', 'C4')),
    Raum       integer unique );
```

```
create table Assistenten
    ( PersNr    integer primary key,
    Name        varchar(30) not null,
    Fachgebietvarchar(30),
    Boss        integer,
    foreign     key (Boss) references Professoren(PersNr)
                    on delete set null);

create table Vorlesungen
    ( VorlNr    integer primary key,
    Titel       varchar(30),
    SWS         integer,
    gelesenVoninteger references Professoren(PersNr)
                    on delete set null);
```

```
create table hoeren
    ( MatrNr     integer references Studenten(MatrNr)
                              on delete cascade,
    VorlNr       integer references Vorlesungen(VorlNr)
                              on delete cascade,
    primary key(MatrNr, VorlNr));

create table voraussetzen
    (Vorgaenger integer references Vorlesungen(VorlNr)
                              on delete cascade,
    Nachfolger   integer references Vorlesungen(VorlNr)
                              on delete cascade,
    primary key (Vorgaenger, Nachfolger));
```

```
create table pruefen
    ( MatrNr integer references Studenten(MatrNr)
                                on delete cascade,
    VorlNr      integer references Vorlesungen(VorlNr),
    PersNr      integer references Professoren(PersNr)
                                on delete set null,
    Note numeric (2,1) check (Note between 0.7 and 5.0),
    primary key (MatrNr, VorlNr));
```

Anmerkungen

Es gibt noch weitere interessante Möglichkeiten bei der Erstellung von Tabellen. z.B.

- Automatische Zuweisung von IDs: create table studenten (matrnr **serial**, ...). Achtung: Syntax serial vs. autoincrement vs. ..... hängt vom Datenbanksystem ab.
- **Default** Werte: create table studenten (matrnr int, vertiefung varchar(20) default 'Informationssysteme', ...)
- Verbindung von default Werten und UDFs: .... vertiefung varchar(20) default getRandVertiefung(), ...

Beispiel:
create table mytable(id serial primary key, vertiefung float default random(), name char(10))
Dann einfach bei insert: insert into mytable(name) values ('Meyer')

Komplexere Konsistenzbedingungen?

```
CREATE TABLE pruefen
(MatrNr integer REFERENCES Studenten on DELETE CASCADE,
VorlNr integer REFERENCES Vorlesungen,
Note numeric(2,1) CHECK (Note BETWEEN 0.7 and 5.0),
PRIMARY KEY (MatrNr, VorlNr),
CONSTRAINT VorherHoeren
CHECK ( EXISTS ( SELECT *
FROM hoeren h
WHERE h.VorlNr = pruefen.VorlNr and
h.MatrNr = pruefen.MatrNr))
);
```

- Bei jeder Änderung und Einfügung wird die check-Klausel ausgewertet.
- Operation wird nur durchgeführt, wenn der check true ergibt.
- Geht leider nicht in aktuellem Postgres 9.5: ERROR: cannot use subquery in check constraint
- nur einfache boolsche Bedingungen im check in Postgres z.B. a < 42

In SQL-92

Primary-Key- und Foreign-Key-Klauseln

- Referentielle Integrität: deklarative Spezifikation unterschiedlicher Reaktionsmöglichkeiten für Wegfall (Löschung, Änderung) eines referenzierten Satzes bzw. Primärschlüssels (CASCADE, ...)

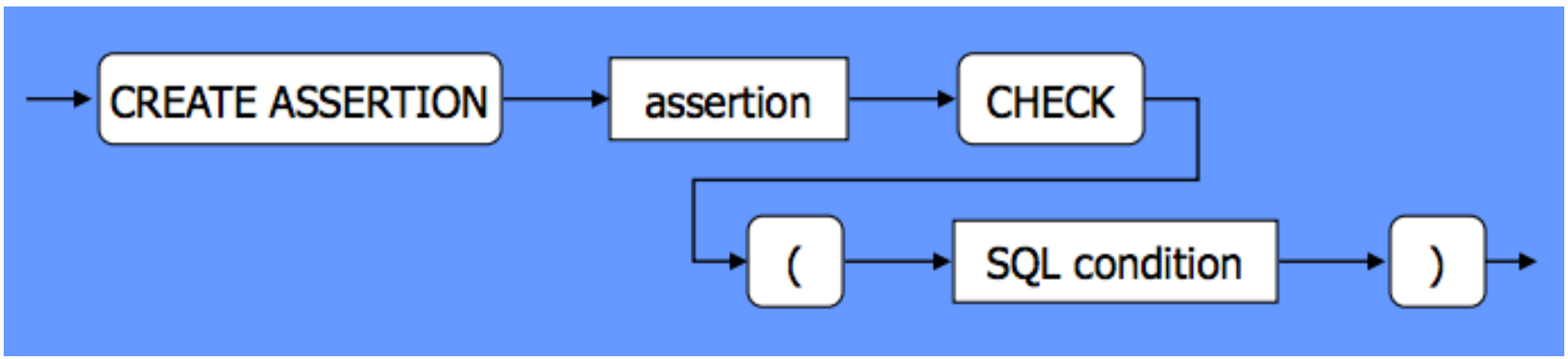
Festlegung von Wertebereichen für Attribute durch Angabe eines Datentyps bzw. Domains

- Optional: Angabe von Default-Werten, Eindeutigkeit (UNIQUE), Nullwerte-Ausschluss (NOT NULL)
- Allgemeine Wertebereichsbeschränkungen über CHECK-Klausel

In SQL-92: Assertions

Assertions: Spezifikation allgemeiner, z.B. tabellenübergreifender Bedingungen

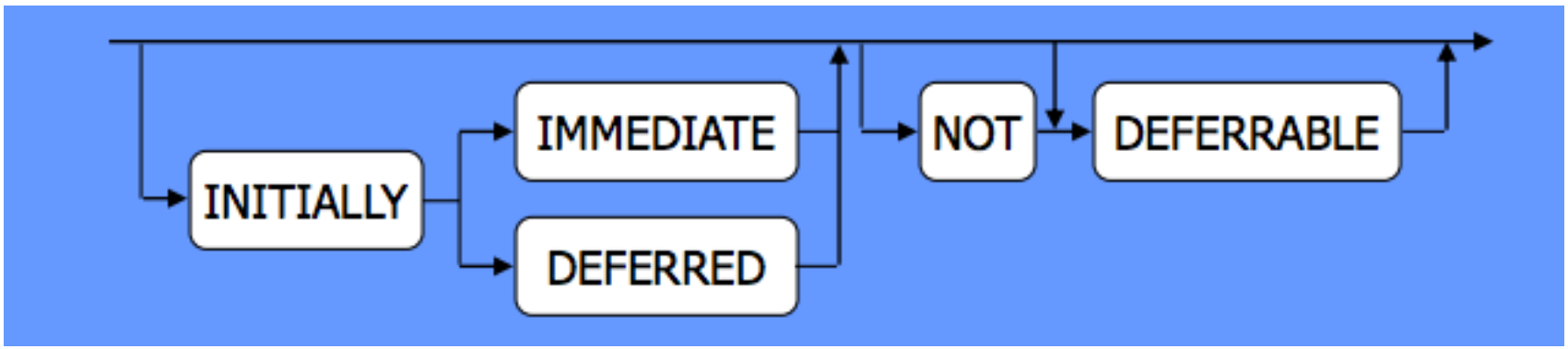
- Beziehen sich typischerweise auf mehrere Relationen
- Lassen sich als eigenständige DB-Objekte definieren
- Erlauben die Verschiebung ihres Überprüfungszeitpunktes
- Syntax der Assertion-Anweisung



In SQL-92: Assertions - IMMEDIATE vs. DEFERRED

Direkte (IMMEDIATE) oder verzögerte (DEFERRED) Überwachung

- **IMMEDIATE**: am Ende der Änderungsoperation (Default)
- **DEFERRED**: am Transaktionsende (COMMIT)



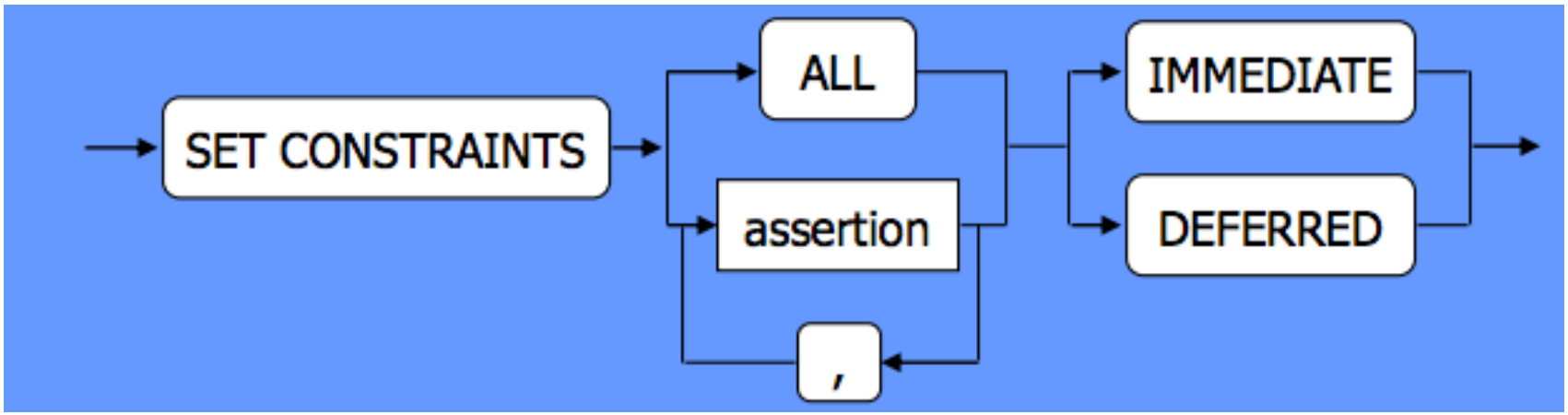
In SQL-92: Assertions - Beispiel

```
CREATE TABLE Pers
(Pnr    int PRIMARY KEY,
Gehalt  int CHECK(Value<500000),
Anr   int NOT NULL
    FOREIGN KEY REFERENCES Abt
    ON DELETE CASCADE
    ...);

CREATE ASSERTION A1
  CHECK (NOT EXISTS
    (SELECT * FROM Abt A
     WHERE A.Anzahl_Angest <>
     (SELECT COUNT(*)
     FROM Pers
     WHERE P.Anr = A.Anr)))
  INITIALLY DEFERRED;
```

In SQL-92: IMMEDIATE oder DEFERRED in Transaktion

- Überprüfung kann innerhalb einer Transaktion gesteuert werden:



- Entweder für alle Constraints/Assertions oder, für benannte Constraints/Assertions selektiv.
- Unter Berücksichtigung von NOT DEFERRABLE
- Wird Postgresql teilweise(!) unterstüzt, aber nur für einfache check constraints.

CHECK OPTION CONSTRAINT in Views

Gegeben folgende Sicht auf die Tabelle professoren

```
CREATE VIEW profsView AS
SELECT *
FROM professoren
WHERE persnr < 2126;
```

Kann das folgende INSERT-Statement ausgeführt werden?

```
INSERT INTO profsView VALUES(2777, 'Michel', 'C4', 444);
```

CHECK OPTION CONSTRAINT in Views (2)

Antwort auf vorherige Frage ist: Ja, es kann ausgeführt werden.

- Das Tupel wird also in die Tabelle professoren eingefügt
- Aber es ist nicht in der View enthalten, da nicht gilt $2777 < 2126$

Um zu vermeiden, dass Tupel eingefügt werden, die nicht in der View enthalten sind, kann die Definition der View wie folgt angepasst werden:

```
CREATE VIEW profsView AS
SELECT *
FROM professoren
WHERE persnr < 2126
WITH CHECK OPTION;
```

CHECK OPTION CONSTRAINT in Views (3)

Alternativ mit **CASCADED CHECK OPTION**, dann werden auch Bedingungen der evtl. Zugrunde liegenden Views überprüft.

CHECK OPTION bzw. CASCADED CHECK OPTION ist von Postgres seit Version 9.4 unterstützt.

Beispiel CASCADED CHECK OPTION CONSTRAINT

```
CREATE VIEW V1 AS SELECT COL1
       FROM T1 WHERE COL1 > 10
```

Es gibt in V1 kein constraint, also funktioniert das folgende Statement:

```
INSERT INTO V1 VALUES (5)
```

View V2 benutzt V1 und hat eine CASCADED CHECK OPTION:

```
CREATE VIEW V2 AS SELECT COL1
 FROM V1 WITH CASCADED CHECK OPTION
```

Also klappt das folgende Insert nicht, da ein CHECK auf V1 forciert wird:

```
INSERT INTO V2 VALUES (5)
```

(Fortsetzung auf nächster Folie)

Aus: `http://www-01.ibm.com/support/knowledgecenter/ssw_ibm_i_61/sqlp/rbafywithcascade.htm`

Fortsetzung des Beispiels

Eine weitere View V3, die von V2 (und somit auch von V1) abhängt:

```
CREATE VIEW V3 AS SELECT COL1
                  FROM V2 WHERE COL1 < 100
```

Das folgende Statement funktioniert nicht, da der Wert 5 ein Problem mit V2 darstellt, da V2 ja die CASCADED CHECK OPTION hat (und daher den Check auf V1 forciert).

```
INSERT INTO V3 VALUES (5)
```

Das folgende Statement funktioniert allerdings:

```
INSERT INTO V3 VALUES (200)
```

Wieso? Es macht keine Probleme bzgl. View V2 (bzgl. V1). Es würde zwar nicht in der View V3 auftauchen, aber das ist kein Problem, weil V3 kein CHECK besitzt.

Database Trigger: Idee

- Assertions/Constraints legen fest was erlaubt ist.
- **Trigger ermöglichen auf bestimmte Ereignisse zu reagieren!**
- Z.B wenn neue Zeilen in eine Tabelle bzw. Sicht eingefügt wurden bzw. werden sollen
- Tritt ein bestimmtes Ereignis auf so wird ein Trigger "gefeuert", d.h. eine vorher festgelegte Aktion (Prozedur bzw. Funktion) ausgeführt.
- Somit können z.B. komplexe Constraints überprüft werden oder Tabellen automatisch angepasst werden

Allgemeines Prinzip: ECA – Event, Condition, Action

Beispiel für Einsatz von Triggern

- Automatisches Nachbestellen, wenn Lagerbestände leerlaufen
- Automatisches Anschreiben von Studenten, wenn creditPoints $<$ vordefinierte Schranke
- Automatische Mahnung für Rechnungen/ablaufende Abos/etc.
- Bibliothek: Anschreiben von Nutzern wenn Ausleihfrist überschritten
- Überwachung von Kontobeständen
  - Obergrenzen für Überweisungen
  - Untergrenzen für Kontostände
- etc.

Ein Teil der Anwendungslogik ist dann in den Triggern definiert.

Beispiel Trigger

- Idee: Jeder neue Student wird automatisch durch einen Eintrag in der Tabelle `hören` für die Vorlesung Logik registriert.
- Also: **AFTER INSERT ON** studenten
- Und für jede neue Zeile einzeln, also **FOR EACH ROW**

```
CREATE TRIGGER pflichtvorlesungLogikTrigger
AFTER INSERT ON  studenten
FOR EACH ROW
EXECUTE PROCEDURE
    pflichtvorlesungLogik_function();
```

Dies ist "nur" der Trigger. Die eigentliche Funktion/Prozedur die beim Eintreffen des Trigger-Ereignisses aufgerufen wird heisst **pflichtvorlesungLogik_function()**.

Beispiel Trigger

```
1  CREATE OR REPLACE
2      FUNCTION pflichtvorlesungLogik_function()
3  RETURNS TRIGGER AS
4  $$
5  BEGIN
6   --- kurze Meldung ausgeben
7   RAISE NOTICE 'Ach, sieh mal an, Student/in %', NEW.name;
8   --- fuege Student/in in Tabelle hoeren ein
9   INSERT INTO hoeren VALUES (New.matrnr, 4052);
10  RETURN NULL;
11 END
12 $$
13 LANGUAGE plpgsql;
```

Database Trigger: Überlegungen

Wann soll ein Trigger ausgelöst werden?

- Zeitpunkt: BEFORE / AFTER / INSTEAD OF
- Auslösende Operation: INSERT / DELETE / UPDATE

Wie spezifiziert man Aktionen?

- Bezug auf verschiedene DB-Zustände erforderlich
- OLD/NEW erlaubt Referenz von alten/neuen Werten

Ist die Trigger-Ausführung vom Zustand der DB abhängig?

- WHEN Bedinging (optional)

Database Trigger: Überlegungen (2)

Wann soll wie verändert werden?

- Pro Tupel (Row) oder pro DB-Operation (Statement): **Trigger-Granulat**?
- mit einer SQL-Anweisung oder mit einer Prozedur aus PL/pgSQL etc?

Existiert das Problem der Terminierung und der Auswertungsreihenfolge?

- Mehrere Trigger-Definitionen pro Tabelle sowie
- mehrere Trigger-Auslösungen pro Ereignis möglich

Ausführungsreihenfolge

- Ein SQL-Änderungs-Statement XYZ wurde abgesetzt
- Frage: **Wann wird welche Art von Trigger ausgeführt?**

FOR EACH STATEMENT:

- Trigger wird ausgeführt für jedes matchende Event
- **einfache** Ausführung des Triggers
- unabhängig von der Anzahl der geänderten Zeilen
- BEFORE Statement: vor allen row-level Triggern
- AFTER Statement: nach allen row-level Triggern

D.h. Reihenfolge ist:

BEFORE Statement → BEFORE Row → AFTER Row → AFTER Statement

Sichtbarkeit von Änderungen

- Ein SQL-Änderungs-Statement XYZ wurde abgesetzt
- Frage: **Welcher Trigger sieht welche Änderung?**

FOR EACH STATEMENT:

- BEFORE: sieht nichts von XYZ
- AFTER: sieht alles

Sichtbarkeit von Änderungen

- Ein SQL-Änderungs-Statement XYZ wurde abgesetzt
- Frage: **Welcher Trigger sieht welche Änderung?**

FOR EACH ROW:

- BEFORE: sieht nichts von XYZ
- AFTER: sieht alles
- Aber: BEFORE sieht u.U. Änderungen anderer BEFORE Trigger
- Reihenfolge der ROW BEFORE Trigger ist aber nicht festgelegt

Beispiel Trigger

- Idee: Jeder neue Student wird automatisch durch einen Eintrag in der Tabelle `hören` für die Vorlesung Logik registriert.
- Also: **AFTER INSERT ON** studenten
- Und für jede neue Zeile einzeln, also **FOR EACH ROW**

```
CREATE TRIGGER pflichtvorlesungLogikTrigger
AFTER INSERT ON  studenten
FOR EACH ROW
EXECUTE PROCEDURE
    pflichtvorlesungLogik_function();
```

Dies ist "nur" der Trigger. Die eigentliche Funktion/Prozedur die beim Eintreffen des Trigger-Ereignisses aufgerufen wird heisst **pflichtvorlesungLogik_function()**.

Beispiel Trigger

```
1  CREATE OR REPLACE
2      FUNCTION pflichtvorlesungLogik_function()
3  RETURNS TRIGGER AS
4  $$
5  BEGIN
6   --- kurze Meldung ausgeben
7   RAISE NOTICE 'Ach, sieh mal an, Student/in %', NEW.name;
8   --- fuege Student/in in Tabelle hoeren ein
9   INSERT INTO hoeren VALUES (NEW.matrnr, 4052);
10  RETURN NULL;
11 END
12 $$
13 LANGUAGE plpgsql;
```

Trigger: Syntax

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF }
    { event [ OR ... ] }
    ON table
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE |
    INITIALLY DEFERRED } ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Trigger: Parameterübergabe an Funktion

- Es macht natürlich wenig Sinn für jede Pflichtvorlesung eine eigene Funktion zu schreiben.
- Besser ist es schonmal die gleiche Funktion zu benutzen und die Vorlesung um die es geht als Parameter zu übergeben, z.B. wie hier für die Vorlesung mit Vorlnr 5001:

```
CREATE TRIGGER pflichtvorlesungTrigger5001
AFTER INSERT ON  Studenten
FOR EACH ROW
EXECUTE PROCEDURE pflichtvorlesung_function(5001);
```

In Postgresql: Trigger-Funktionen haben KEINE Parameter in ihrer Definition. Aber, es können Parameter übergeben werden an die Trigger-Funktion. Dies geschieht über die Variable TG_ARGV

Beispiel Trigger: Parameterübergabe via TG_ARGV

```
1  CREATE OR REPLACE FUNCTION pflichtvorlesung_function()
2  RETURNS TRIGGER AS
3  $$
4  DECLARE
5      vorlnr_param int;
6      titel_param varchar;
7  BEGIN
8    --- PflichtVL wird per TG_ARGV uebergeben
9    vorlnr_param :=  TG_ARGV[0];
10   --- Fuer NOTICE ist es huebscher  Titel der VL zu haben
11   SELECT titel into titel_param FROM vorlesungen where
       vorlnr=vorlnr_param;
12   RAISE NOTICE 'Ach,  Student/in %', NEW.name ;
13   RAISE NOTICE 'sollte auf jeden Fall die VL % (%) hoeren
       ', titel_param, vorlnr_param;
14   INSERT INTO hoeren VALUES (New.matrnr, vorlnr_param);
15   RETURN NULL;
16 END $$ LANGUAGE plpgsql;
```

Beispiel Trigger: Aufruf

```
INSERT INTO studenten VALUES (29000,'Michel','1');
```

```
NOTICE: Ach,  Student/in Michel
NOTICE: sollte auf jeden Fall die VL Grundzuege (5001) hoeren
```

Hinweis zum "Herumspielen/Debuggen"

Damit die Tabelle, in die eingefügt wird, nicht zugemüllt wird bietet es sich an hier eine Transaktion mit rollback() zu benutzen:

```
BEGIN TRANSACTION;
INSERT INTO studenten VALUES (29000,'Michel','1');

ROLLBACK;
```

Weiteres Beispiel

Tabelle mit Informationen über Angestellte:

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);
```

Der Eintrag last_date soll den Zeitpunkt der letzten Änderung an der jeweiligen Zeile angeben. last_user soll entsprechend den Namen des Benutzers enthalten, der diese Zeile eingefügt bzw. aktualisiert hat.

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Aus:

https://www.postgresql.org/docs/9.5/static/plpgsql-trigger.html

Weiteres Beispiel (2)

https://www.postgresql.org/docs/9.5/static/plpgsql-trigger.html

```
1  CREATE FUNCTION emp_stamp() RETURNS trigger AS $$
2  BEGIN
3   -- Name und Gehalt sollten nicht NULL sein
4   IF NEW.empname IS NULL THEN
5       RAISE EXCEPTION 'empname cannot be null';
6   END IF;
7   IF NEW.salary IS NULL THEN
8       RAISE EXCEPTION '% cannot have null salary', NEW.
      empname;
9   END IF;
10
11  -- Vermutlich wird niemand fuer ein negatives Gehalt
      Arbeiten wollen
12  IF NEW.salary < 0 THEN
13      RAISE EXCEPTION '% cannot have a negative salary',
      NEW.empname;
14  END IF;
```

Weiteres Beispiel (2)

```
15  -- Merken wer diesen Eintrag geaendert hat und wann
16  NEW.last_date := current_timestamp;
17  NEW.last_user := current_user;
18  RETURN NEW;
19 END;
20 $$ LANGUAGE plpgsql;
21
```

Rückgabewerte der Trigger-Funktion: In Postgresql

In Postgresql gibt eine Triggerfunktion entweder NULL zurück oder ein Tupel, das dem Schema der Relation auf die der Trigger definiert ist entspricht.

- Return value von AFTER ROW Triggern sowie für BEFOR oder AFTER STATEMENT Triggern werden immer ignoriert; der Wert kann also auch NULL sein.
- Für BEFORE ROW Trigger wird der Rückgabewert NULL so gedeutet, dass keine nachfolgenden Trigger (für diese Zeile) mehr ausgeführt werden. Wird ein nicht-NULL Wert zurückgegeben, so arbeiten Trigger, die danach aufgerufen werden mit diesem Wert.
- Vorsicht bei DELETE Triggern bei return NULL (besser RETURN OLD; wird sowieso ignoriert)

Ausführliche Beschreibung unter: `https://www.postgresql.org/docs/9.5/static/plpgsql-trigger.html`

Row-Trigger: Referenzen auf alte bzw. neue Versionen der Zeilen

- Für Trigger, die pro Zeile (Row) ausgeführt werden, sogenannte Row-Trigger kann direkt auf die betroffene Zeile zugegriffen werden.
- Bei INSERT natürlich nur auf die neue Zeile. Schlüsselwort NEW.
- Bei UPDATE auf NEW sowie OLD.
- Bei DELETE nur auf OLD

Referenzen Definieren

- Die jeweiligen alten bzw. neuen Versionen können direkt referenziert werden oder
- es können via REFERENCES Alias eingeführt werden (Im SQL Standard, nicht in PG)

Für Row-Trigger sind die o.g. Referenzen gültig unabhängig davon ob der Trigger BEFORE oder AFTER der Operation ausgeführt wird.

Trigger in Postgresql

- Nicht komplett kompatibel zum SQL Standard
- Z.B. kann man in PG keinen Alias OLD bzw. NEW einführen
- und generell für STATEMENT Trigger nicht auf OLD_TABLE bzw. NEW_TABLE zugreifen
- SQL Standard sagt Trigger sollten in Reihenfolge der Erzeugung ausgeführt werden, Postgresql aber sortiert nach Namen
- CREATE CONSTRAINT TRIGGER ist eine Erweiterung in Postgres
  - In Zusammenhang mit **create table** verwendbar
  - Schließt den Kreis zu komplexeren check constraints

`https://www.postgresql.org/docs/9.5/static/sql-createtrigger.html`

Gültige Kombinationen

Granulat	Aktivierungs-zeit	Triggernde Operation	Übergangs-variablen erlaubt	Übergangstabellen erlaubt
ROW	BEFORE	INSERT	NEW	NONE
ROW	BEFORE	UPDATE	OLD, NEW	NONE
ROW	BEFORE	DELETE	OLD	NONE
ROW	AFTER	INSERT	NEW	NEW_TABLE
ROW	AFTER	UPDATE	OLD, NEW	OLD_TABLE, NEW_TABLE
ROW	AFTER	DELETE	OLD	OLD_TABLE
STATEMENT	BEFORE	INSERT	NONE	NONE
STATEMENT	BEFORE	UPDATE	NONE	NONE
STATEMENT	BEFORE	DELETE	NONE	NONE
STATEMENT	AFTER	INSERT	NONE	NEW_TABLE
STATEMENT	AFTER	UPDATE	NONE	OLD_TABLE, NEW_TABLE
STATEMENT	AFTER	DELETE	NONE	OLD_TABLE

Probleme mit Triggern

Mögliche rekursive Aufrufe von Triggern

- Änderung auf Tabelle A feuert Trigger Tr1
- Tr1 ändert Tabelle B
- Änderung auf Tabelle B feuert Trigger Tr2
- Tr2 ändert Tabelle A ...

Mögliches verwirrendes Geflecht von Triggern in der Praxis

- Was passiert bei einer Änderungsoperation wirklich?
- Terminiert die Rekursion der Trigger?
- Führen die Trigger Inkonsistenzen in die DB ein?

Deshalb: Vorsicht beim Einsatz von Triggern! Insbesondere bei Triggern, die selbst Daten einfügen.

Wird in Postgresql nicht überprüft:

"It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios." (aus der Postgres Dokumentation über Trigger).