



Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Wiederholung: Stored Procedures/UDFs

Manchmal ist es aber sinnvoll, Teile der Anwendung direkt im DBMS auszuführen und nicht via einzelnen SQL Statements.

Vorteile

- Daten müssen nicht erst auf dem DBMS zur Anwendung gebracht werden (und umgekehrt)
- Höhere Performanz
- Code kann wiederverwendet werden (zwischen Anwendungen)

Nachteile

- Etwas aufwendiger zu erstellen.
- Debugging schwieriger.

Wiederholung: Tabellen als Rückgabewerte: Table Functions

```
CREATE FUNCTION getProfs(int)
RETURNS TABLE(persnr int) AS $$
  SELECT persnr from professoren p
  WHERE p.persnr < $1 ;
$$ LANGUAGE SQL;
```

```
select * from getProfs(2130);
```

```
persnr
-----
  2125
  2126
  2127
(3 rows)
```

Wiederholung: PL/pgSQL: SELECT INTO

SQL Anfragen, die nur eine Zeile zurück liefern können direkt in Variablen eingelesen werden, z.B.

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
```

Falls mehrere Ergebnisse geliefert werden wird die erste Zeile benutzt.
Durch die Angabe von STRICT, also

```
SELECT * INTO STRICT myrec FROM emp  
WHERE empname = myname;
```

wird darauf geachtet, dass es nur genau ein Ergebnis gibt (ansonsten wird eine Exception geworfen).

Siehe **EXECUTE** für dynamische Anfragen und **PERFORM** für Anfragen ohne Ergebnisse zu berücksichtigen:

<http://www.postgresql.org/docs/9.3/static/plpgsql-statements.html>

Wiederholung: PL/pgSQL: Kontrollstrukturen und SQL Anfragen

Über Anfrageergebnisse iterieren

```
CREATE OR REPLACE FUNCTION testit() RETURNS int as
$$
DECLARE
    myprofs RECORD;
    myint int = 0;
BEGIN
FOR myprofs in SELECT * FROM professoren
                    WHERE persnr < 2130
LOOP
    myint = myprofs.persnr + myint;
END LOOP;
return myint;
END;
$$ LANGUAGE plpgsql;
```

PL/pgSQL: IF Statement

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    — dann muss die Zahl wohl NULL sein ...
    result := 'NULL';
END IF;
```

Siehe auch [CASE](#) statements.

PL/pgSQL: Weitere Befehle - RAISE NOTICE

Zum Ausgeben von Meldungen/Warnungen oder einfach zur zum Debuggen Ihreres PL/pgSQL Codes:

```
RAISE NOTICE 'Parameter x = % und y = %', x, y
```

PL/pgSQL: Exception Handling

Syntax

```
....  
EXCEPTION  
    WHEN condition [ OR condition ... ] THEN  
    ....
```

Beispiel

```
BEGIN  
    x := x + 1;  
    y := x / 0;  
EXCEPTION  
    WHEN division_by_zero THEN  
        RAISE NOTICE 'caught division_by_zero';  
        RETURN x;  
    — bzw. äquivalent: WHEN SQLSTATE '22012' THEN  
END;
```


Zusammenfassung UDFs bzw. PL/pgSQL

- Die Implementierung von Teilen der Anwendungslogik direkt im Datenbanksystem kann einige Vorteile haben. Z.B. Wiederverwendbarkeit von Code und dass Daten nicht bewegt werden müssen.
- Realisiert im DBMS durch Stored Procedures bzw. User Defined Functions (UDFs)
- Basierend auf prozeduraler Sprache (PSM=Persistent Stored Modules)
- Haben uns PL/pgSQL als Beispiel genauer angeschaut

Ausblick

- Integritätskontrolle im DBMS durch Trigger: Dort werden in PL/pgSQL Prozeduren geschrieben, die beim Eintreffen eines Ereignisses ausgeführt werden.

Integritätskontrolle

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

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)

Teile des Univeritätsschema mit Integritätsbedingungen

```
CREATE TABLE hoeren
(MatrnNr integer REFERENCES Studenten(MatrnNr)
ON DELETE CASCADE,
VorlNr integer REFERENCES Vorlesungen(VorlNr)
ON DELETE CASCADE,
PRIMARY KEY(MatrnNr, VorlNr));
```

```
CREATE TABLE voraussetzen
(Vorgaenger integer REFERENCES Vorlesungen(VorlNr)
ON DELETE CASCADE,
Nachfolger integer REFERENCES Vorlesungen(VorlNr)
ON DELETE CASCADE,
PRIMARY KEY(Vorgaenger, Nachfolger));
```

Teile des Univeritätsschema mit Integritätsbedingungen (2)

```
CREATE TABLE pruefen
(MatrnNr integer REFERENCES Studenten(MatrnNr)
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 (MatrnNr, VorlNr));
```


Komplexere Konsistenzbedingungen?

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

- Bei jeder Änderung und Einfügung wird die check-Klausel ausgewertet.
- Operation wird nur durchgeführt, wenn der check true ergibt.
- Geht leider nicht im aktuellen Postgres 9.3: **ERROR: cannot use subquery in check constraint**
- nur einfache boolsche Bedingungen im check in Postgres wie 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, ...)

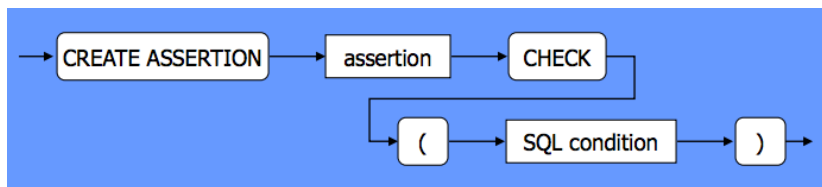
Festlegung von Wertebereichen für Attribute durch Angabe eines Datentyps bzw. Domains

- 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

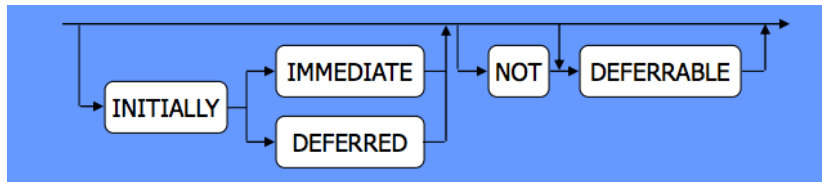
- Beziehen sich typischerweise auf mehrere Relationen
- Lassen sich als eigenständige DB-Objekte definieren
- Erlauben die Verschiebung ihres Überprüfungszeitpunktes
- Syntax der Assertion-Anweisung



In SQL-92: Assertions - IMMEDIATE vs. DEFERRED

Direkte (IMMEDIATE) oder verzögerte (DEFERRED) Überwachung

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



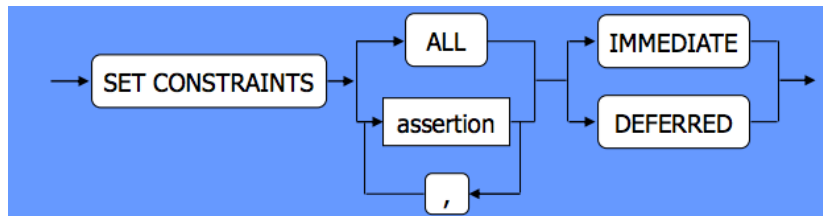
In SQL-92: Assertions - Beispiel

```
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 <math><<math>
    (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(!) unterstützt, 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);
```

Anmerkung/Richtigstellung zu Aktualisierbarkeit von Views in Postgresql.
In der Tat unterstützt Postgresql ab Version 9.3 Updates auf Views gemäß SQL-92 Standard.

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.

Und eine (weitere/obligatorische) Anmerkung zur Kompatibilität: CHECK OPTION bzw. CASCADED CHECK OPTION wird in Postgres 9.3 nicht 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 Bedingung (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 den Titel der VL zu
11    haben
12    SELECT titel into titel_param FROM vorlesungen where
13        vorlnr=vorlnr_param;
14    RAISE NOTICE 'Ach, sieh mal an, Student/in %', NEW.name ;
15    RAISE NOTICE 'sollte auf jeden Fall die VL % (%) hoeren ',
16        titel_param , vorlnr_param;
17    INSERT INTO hoeren VALUES (New.matnr, vorlnr_param);
18    RETURN NULL;
19 END;
```

Beispiel Trigger: Aufruf

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

NOTICE: Ach, sieh mal an, 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:

<http://www.postgresql.org/docs/9.3/static/plpgsql-trigger.html>

Weiteres Beispiel (2)

<http://www.postgresql.org/docs/9.3/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 geändert 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 BEFORE 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: [http:](http://www.postgresql.org/docs/9.3/static/plpgsql-trigger.html)

[//www.postgresql.org/docs/9.3/static/plpgsql-trigger.html](http://www.postgresql.org/docs/9.3/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 NEW

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

<http://www.postgresql.org/docs/9.3/static/sql-createtrigger.html>

Gültige Kombinationen

Granulat	Aktivierungszeit	Triggernde Operation	Übergangsvariablen 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).