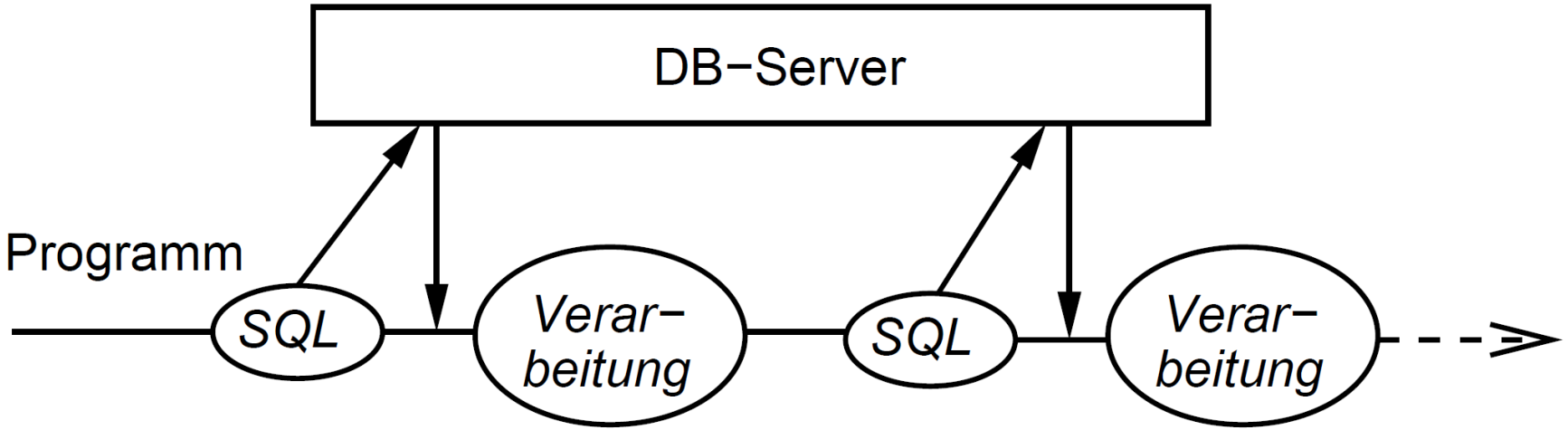
Datenbanksysteme

Kap 5: Server-seitige DB-Programmierung

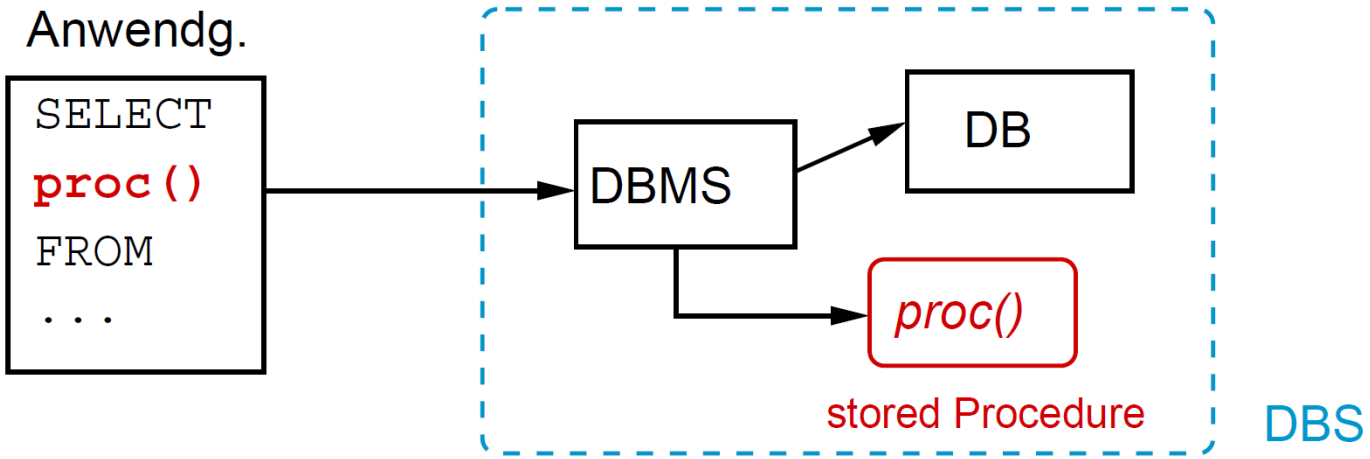
Rückblick: Client-seitige DB-Programmierung

- Datenbankzugriff aus Client-Programm
  - Programm holt Daten per SQL
  - Abhängig von Verarbeitungslogik werden weitere SQL-Statements abgesetzt usw.
  - Verarbeitungslogik kann fest programmiert sein (z.B. Praktikum 2) oder interaktiv vom Anwender bestimmt werden (Extremfall: SQL-Prompt)



Idee serverseitige Programmierung

- Verlagere Datenmanipulation mit fester Ablauflogik vom Client auf den Server
  - Programme werden als stored Procedures im DBS hinterlegt und vom Clientprogramm per SQL-Befehl gestartet



- Vorteile
  - Stored Procedures stehen in jedem Client zur Verfügung (sogar im SQL-Prompt)
  - Bei Änderung der Ablauflogik müssen Clients nicht angepasst werden, sondern nur zentrale Prozedur

Komponenten serverseitiger Programmierung

- Stored Procedures
  - Im Server hinterlegte Programme
  - SQL nicht Turing-vollständig → Procedures können i. allg. nicht in reinem SQL sein (Host Language oder PL/SQL)
- Prozedurale SQL-Erweiterungen
  - Ermöglichen direkte Erzeugung von Prozeduren mit SQL
  - Quasistandard: PL/SQL von Oracle
- Trigger
  - Auslösen von Stored Procedures beim Eintreten bestimmter Ereignisse, z.B. Update einer bestimmten Tabelle
  - "Aktive" Datenbankobjekte, die nicht direkt vom Anwender angesprochen werden

"Standards" der Server-seitigen Programmierung

- PL/SQL
  - Prozedurale SQL-Erweiterung von Oracle
  - Vollwertige moderne prozedurale Programmiersprache
    - Funktionen mit Default-Args und Überladen, Exceptions,...
  - Vorbild für PL/pgSQL von PostgreSQL
- PSM
  - Persistent Stored Modules (PSM) 1996 in ANSI SQL-Standard aufgenommen; Bestandteil von SQL3 (1999)
  - Spezifiziert drei Aspekte
    - Definition und Aufruf von Prozeduren und Funktionen
    - Zusammenfassen von Funktionen zu Modulen
    - Prozedurale SQL-Erweiterung
  - Wegen zu PL/SQL inkompatibler Syntax (noch?) kaum umgesetzt

Definition und Aufruf von Stored Procedures

- SQL3 unterscheidet zwischen Prozedur und Funktion
  - Anlage mit `CREATE PROCEDURE` bzw. `CREATE FUNCTION`
  - Aufruf Prozedur mit SQL-Befehl `CALL <procname>`
  - Aufruf Funktion im Rahmen von `SELECT`-Statement
    - Beispiel: `SELECT bruttofunc(preis) FROM produkt;`
- PostgreSQL macht diese Unterscheidung nicht
  - expliziter Aufruf nur über `SELECT` möglich → explizit aufrufbare Funktion muss Rückgabewert haben
  - Funktionen ohne Rückgabewert haben Rückgabetyp `TRIGGER`
    - können nur implizit vom DBS aufgerufen werden (z.B. über Trigger)

Beispiel: Berechnung

```
DROP FUNCTION bruttofunc(NUMERIC);
-- berechne Bruttobetrag incl. Mwst
CREATE FUNCTION bruttofunc(NUMERIC)
  RETURNS NUMERIC AS '
    SELECT $1 * CAST(1.16 AS NUMERIC);
  ' LANGUAGE 'SQL';
```

- Bemerkungen:
  - Überladung möglich → Argumente beim `DROP` angeben
  - Argumente referenzierbar mit $1, $2 etc.
  - Rückgabewert = Ergebnis letztes `SELECT`-Statement
  - Funktion ist mit reinem SQL implementiert (kein PL/SQL)
  - Implementierungssprache im Parameter `LANGUAGE` angegeben
    - auch andere Sprachen möglich (plpgsql, C, plperl, pltcl, plpython)

Beispiel: Operationen

- Funktionen können nicht nur zum Berechnen, sondern auch für Operationen verwendet werden:

```
-- Abbuchung in Tabelle Konto durchführen
CREATE FUNCTION abbuchung(VARCHAR, NUMERIC)
  RETURNS NUMERIC AS '
    UPDATE konto SET stand = stand - $2
      WHERE nr = $1;
    SELECT stand FROM konto WHERE nr = $1;
  ' LANGUAGE 'SQL';
```

- Bemerkung
  - Letztes SELECT nötig, da Funktion Wert zurückgeben muss
  - Könnte aber auch durch triviales SELECT ersetzt werden, z.B. `SELECT '1';`

Andere Implementierungssprachen

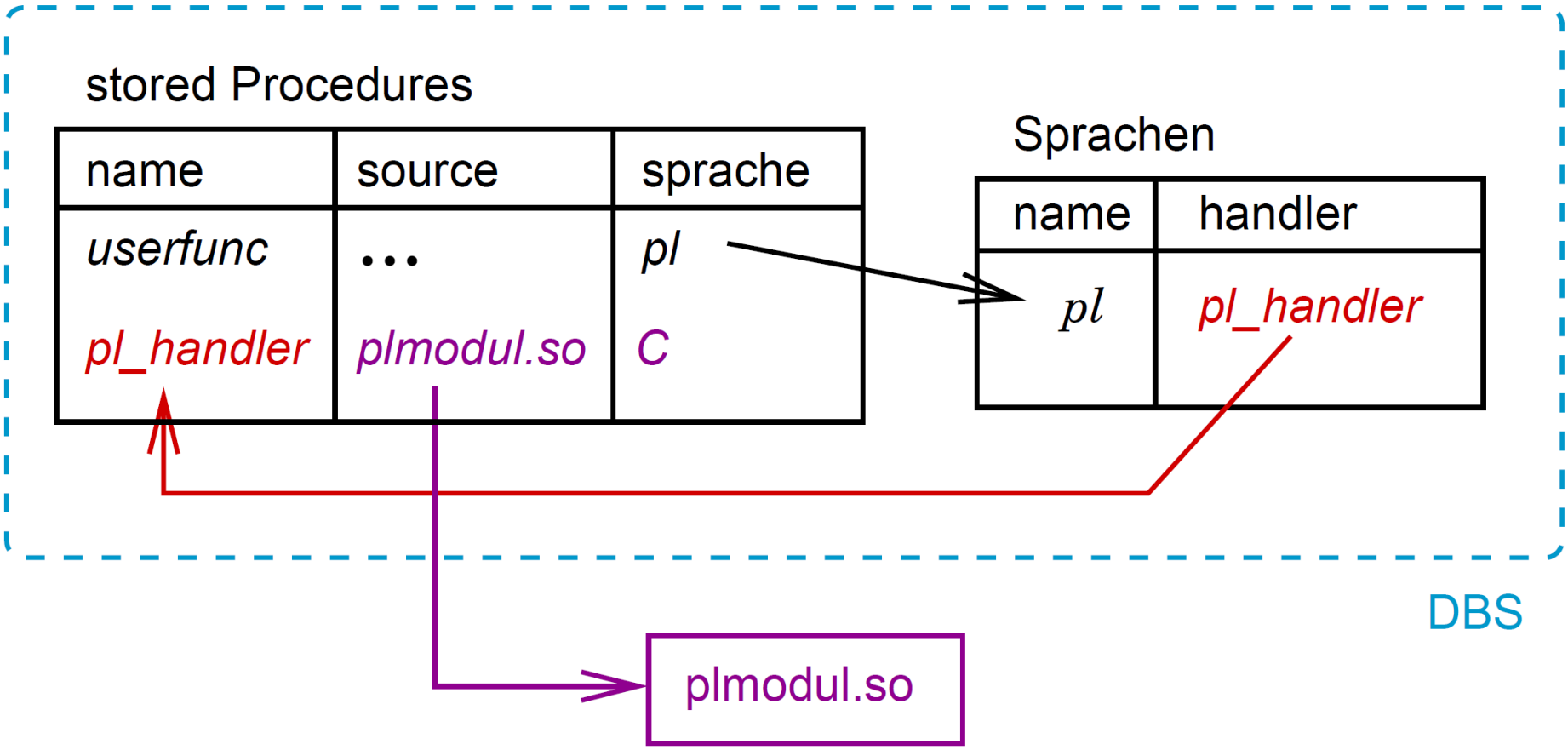
- PostgreSQL unterstützt auch C-Funktionen
  - Funktion muss in Shared-Library (*.so) bereitgestellt werden
  - Shared-Library wird auf DB-Server abgelegt
  - Suchpfad für Shared-Library ist konfigurierbar
  - Parameterübergabe über spezielle libpq-Makros
- SQL Funktionsdeklaration:
  -
```
CREATE FUNCTION bruttofunc(numeric)
   RETURNS numeric AS 'bruttofunc.so'
   LANGUAGE 'C';
```
- Nachteile:
  - Zugriff auf Betriebssystem erforderlich
    - Im allgemeinen nur durch DBA möglich
    - Plattformabhängig
  - DBS kann nicht optimieren

Prozedurales SQL

- SQL nur begrenzte Möglichkeiten
  - Für "inline" Definition von Funktionen ist prozedurale SQL-Erweiterung nötig

- Ansätze
  - feste Implementierung von PL/SQL im DBS
    - Von Oracle gewählte Lösung
    - PL/SQL immer verfügbar (auch außerhalb von Funktionen!)
  - Framework zum Bereitstellen von Sprachen
    - Von PostgreSQL gewählte Lösung
    - Sprachen müssen separat ins DBS eingebunden werden
    - Sprache in Funktionsdefinition angeben
    - Beliebig erweiterbarer Ansatz

Sprachen-Framework in PostgreSQL



Installation Sprache in PostgreSQL (durch DBA)

- Compilieren und Bereitstellen Objectfile für den Language Handler
  - ggf. schon vorinstalliert, z.B. bei plpgsql
- Deklaration Handler Funktion
  - `CREATE FUNCTION handler_function_name() RETURNS LANGUAGE HANDLER AS 'path-to-shared-object' LANGUAGE C;`
- Deklaration der Sprache
  - `CREATE [TRUSTED] LANGUAGE language-name HANDLER handler_function_name;`
- Hinweise:
  - für mitgelieferte Sprachen (plpgsql, plperl, ...) kann Script `createlang` verwendet werden, z.B. `createlang plpgsql template1`
  - bei Installation in `template1` werden Sprachen an (danach!) angelegte Datenbanken vererbt

Sicherheitsaspekte

- Sprachen können Aufruf von System-Kommandos ermöglichen → ggf. Sicherheitsproblem
  - Beispiel: Prozedur *xp_cmd_shell* in MS SQL-Server
  - Jeder DB-User darf beliebige Systemkommandos mit der Userid ausführen, unter der das DBS läuft

- Lösungsansätze:
  - Lasse Systemcalls nicht zu in Sprache ("trusted Language") → Sprache darf von jedem benutzt werden
  - Lasse "untrusted" Languages nur für spezielle User zu
  - In PostgreSQL Parameter `trusted` bei `create language`
    - Benutzung "untrusted" Language erfordert DBA-Rechte

Struktur von PL/SQL und PL/pgSQL

- Allgemeine Struktur ist blockorientiert:

```
[ DECLARE
    declarations ]
BEGIN
    statements
END;
```

- Statements werden mit Semikolon (;) angeschlossen
- Jedes Statement kann selber wieder Block sein
- Deklarationen gelten nur im jeweiligen Block
- BEGIN nicht zu verwechseln mit Transaktionsstart:
  - PostgreSQL erlaubt keine verschachtelten Transaktionen und damit auch keine Transaktionen innerhalb von Funktionen
  - Oracle kennt kein BEGIN WORK (nur impliziter Transaktionsbeginn)

Beispiel

- Beispielfunktion in SQL:

```
-- berechne Bruttobetrag incl. Mwst
CREATE FUNCTION bruttofunc(NUMERIC)
  RETURNS numeric AS '
    SELECT $1 * CAST(1.16 AS NUMERIC);
  ' LANGUAGE 'SQL';
```

- Dieselbe Funktion in PL/pgSQL:

```
CREATE FUNCTION bruttofunc(numeric)
  RETURNS numeric AS '
    DECLARE
      res NUMERIC;
    BEGIN
      res := $1 * CAST(1.16 AS NUMERIC);
      RETURN res;
    END;
  ' LANGUAGE 'plpgsql';
```

Mögliche Deklarationen

```
-- normaler SQL-Datentyp
name VARCHAR(30);

-- Vorbelegung jedesmal, wenn Block aufgerufen
menge INT DEFAULT 0; /*oder: menge INT := 0;*/

-- Datentyp von Tabellenattribut übernehmen
preis produkt.preis%TYPE;

-- Aliasname für Funktionsparameter
arg1 ALIAS FOR $1;

-- zusammengesetzter Datentyp (Tabellentupel)
prod produkt%ROWTYPE;

-- Platzhalter für SELECT-Ergebnis
-- (d.h. ROWTYPE mit beliebiger Struktur)
rec RECORD;
```

Kontrollstrukturen

- Verzweigungen
  - `if - then - [else -] end if;`

- Loops
  - sowohl while- als auch for-Loops:

```
WHILE bedingung LOOP
  anweisungen
END LOOP;
```

```
FOR var IN start .. ende LOOP
  anweisungen
END LOOP;
```

  - Abbruch aus Schleife mit exit
  - Auch Loops über SELECT-Ergebnisse möglich

```
FOR rec IN SELECT * FROM produkt LOOP
  IF (rec.preis < 5) THEN
     zaehler := zaehler + 1;
  END IF;
END LOOP;
```

SQL-Statements in prozeduralem SQL

- UPDATE, INSERT, DELETE direkt formulierbar
- SELECT ist komplizierter:
  - Was ist, wenn mehr als ein Tupel zurückliefert wird?
  - Wie wird erkannt, ob überhaupt Ergebnis gefunden?

- Single-Row Select
  - Kann mit SELECT INTO erfolgen:

```
SELECT max(preis) INTO maxpreis FROM produkt;
```

  - Komplettes Tupel kann in `record` oder `rowtype` Variable eingelesen werden
    - Attributwerte ansprechbar mit `rec.att`

Multi-Row Select

- einfache Variante (nur Postgres): Select in For-Loop
  -
```
DECLARE
  rec RECORD;
BEGIN
  FOR rec IN SELECT * FROM produkt LOOP
    /* Verarbeitung */
  END LOOP;
END;
```
- komplizierte Variante (Postgres und Oracle): Cursor
  -
```
DECLARE
  cur CURSOR IS SELECT * FROM produkt;
BEGIN
  OPEN cur;
  LOOP
    FETCH cur INTO variablelist;
    EXIT WHEN cur%NOTFOUND; /*Postgres: EXIT WHEN NOT FOUND;*/
    /* Verarbeitung */
  END LOOP;
  CLOSE cur;
END;
```

Überprüfung des SELECT-Ergebnisses

- Postgres
  - Abfragen globale boolesche Variable `found`
  - True, wenn letztes SELECT ein nicht-leeres Ergebnis lieferte
- Oracle
  - Wenn `SELECT INTO` nichts liefert, wird Exception vom Typ *`no_data_found`* geworfen
  - Bei `fetch into` Cursorattribut*`%NOTFOUND`* abfragen

Dynamische Statements

- Statements, die erst zur Laufzeit zusammengesetzt werden, können mit `EXECUTE` ausgeführt werden
- `EXECUTE` auch dann nötig, wenn Statement Tabellen referenziert, deren OID zur Compilezeit noch nicht bekannt ist (trifft z.B. auf DDL-Statements zu)
- Beispiel:

```
EXECUTE format(
    'SELECT count(*) FROM %I
     WHERE inserted_by = $1 AND inserted <= $2',
                  tabname)
    INTO c
    USING checked_user, checked_date;
```

Fehlerbehandlung

- Erfolgt grundsätzlich über Exceptions
- Postgres
  - Stark eingeschränktes Fehlerhandling: Exceptions können zwar mit `raise exception` geworfen, aber nicht gefangen werden
  - Schlägt SQL-Statement fehl, wird aktuelle Transaktion mit rollback abgebrochen
- Oracle
  - PL/SQL Blöcke haben zusätzlichen exception Abschnitt, in dem auf Exceptions je nach Typ verschieden reagiert werden kann

Komplettes Beispiel

Mwst

mwst	gueltigab#
14.00	01.01.1990
16.00	01.04.1997

Lieferung

lnr#	produkt	netto	datum
1	Pritt	50.12	01.12.1992
2	Uhu	82.50	01.12.1999

- Funktion zur Mehrwertsteuerberechnung
  - zwei Argumente: Nettobetrag, Datum
  - sucht zu Datum passenden Mwst-Satz und berechnet Bruttobetrag
  - wenn zu Datum kein passender Mwst-Satz hinterlegt → Fehler (Alternative: Rückgabe von NULL)
- Siehe plsqldemo.zip auf DBS-Moodleseite

Was ist ein Trigger?

- Trigger verknüpfen ein Ereignis in einer Tabelle mit bestimmten Aktionen
  - Auslösendes Ereignis kann INSERT, UPDATE oder DELETE sein;
  - Ist immer an genau eine Tabelle gebunden
  - Ausgelöste Aktion kann beliebige Stored Procedure sein (kann also auch andere Tabellen betreffen)
  - Auch als Event-Condition-Action Rules (ECA) bezeichnet
- Trigger können von Anwendern nicht direkt angestossen werden (nur indirekt durch Ereignis)
- Von meisten DBS unterstützt und in SQL3 enthalten, aber zahlreiche Unterschiede im Detail

Wozu sind Trigger gut?

- Automatisierung Ablauflogik
  - Abläufe können im DB-Server hinterlegt werden, ohne dass Clientprogramm spezielle Funktionen aufrufen muss
  - Abläufe können unabhängig vom Client erzwungen werden
- Komplexe Integrity Constraints
  - Variante 1: erzeuge Fehler (Exception) bei Integritätsverletzung
  - Variante 2: korrigiere fehlerhafte Eingabe automatisch
- Berechnung redundanter Werte
  - Aus Performancegründen oft keine Redundanzfreiheit
  - Update-Anomalien können durch Trigger aufgelöst werden
- Wichtig: nur dosiert und mit Bedacht einsetzen!

Problematische Eigenschaften von Triggern

- Strukturierung
  - Es fehlen z.Zt. Abstraktionsmechanismen, um Trigger zu logischen Einheiten zusammenzufassen
- Terminierung
  - Operationen in Triggerfunktionen können andere Trigger (evtl. auch sich selber!) auslösen
  - Terminiert diese Triggerkette?
  - Frage ist für beliebige Kombinationen unentscheidbar (vgl. THI)
- Konfluenz
  - Dasselbe Ereignis kann mehrere Trigger parallel auslösen
  - Ist das Ergebnis unabhängig von der Abarbeitungsreihenfolge?
  - Auch diese Frage ist im allg. unentscheidbar

Anlegen eines Triggers

- Anlegen eines Triggers (PostgreSQL)
  - `CREATE TRIGGER trigger [ BEFORE | AFTER ] event ON relation FOR EACH [ ROW | STATEMENT ] EXECUTE PROCEDURE procedure();`
- Auslösendes Ereignis (`event`) kann `INSERT`, `UPDATE` oder `DELETE` sein; auch Kombinationen mit `OR` möglich
- Triggerfunktion kann vor (`BEFORE`) oder nach (`AFTER`) dem auslösenden Ereignis aufgerufen werden
- Ausführen für jede vom event betroffene Zeile (`FOR EACH ROW`) oder nur einmal pro gesamtes Statement (`FOR EACH STATEMENT`)

Triggerfunktion

- Mit einem Trigger verknüpfte Funktion sieht so aus:

```
CREATE FUNCTION triggerfunc() RETURNS TRIGGER
  AS ' ... ' LANGUAGE 'plpgsql';
```

  - Funktion als Triggerfunktion markiert (Rückgabewert `TRIGGER`)
  - Innerhalb der Funktion enthalten spezielle Variablen Informationen über auslösendes Ereignis und Zustand
  - Variablen sind DBS-spezisch. Bei Postgres:

Variable	Typ	Bedeutung
`OLD`	`RECORD`	Datensatz vor Ausführung Ereignis
`NEW`	`RECORD`	Datensatz nach Ausführung Ereignis
`TG_NAME`	`NAME`	Name auslösender Trigger
`TG_RELNAME`	`NAME`	Name auslösende Tabelle
`TG_OP`	`TEXT`	Art auslösendes Ereignis (*insert*,...)

Beispiel: Protokollierung letztes Update

- Tabelle habe Attribut `lastchange` für Zeitpunkt des letzten Updates
- Definition der Triggerfunktion:

```
CREATE FUNCTION changelog() RETURNS TRIGGER AS '
  begin
    new.lastchange := current_timestamp;
    return new;
  end;
' LANGUAGE 'plpgsql';
```

- Definition des eigentlichen Triggers:

```
CREATE TRIGGER tg_mytable
  BEFORE UPDATE ON mytable
  FOR EACH ROW EXECUTE PROCEDURE changelog();
```

Trigger für Integrity Constraints

- Begrenzter Leistungsumfang eingebauter Constraints
  - NOT NULL, CHECK bezieht sich nur auf aktuelles Tupel (aber: flexibler, wenn CHECK Subselects zulässt)
  - UNIQUE, PRIMARY KEY, FOREIGN KEY prüfen Vorkommen in Relation
- SQL3 Assertion hat sich nicht durchgesetzt
  - Sehr schwierig zu implementieren
  - Trigger sind flexibler, weil zusätzliche Operationen möglich
- Trigger können beliebige Bedingungen prüfen
  - durch PL/SQL nicht auf relationale Algebra beschränkt

Komplettes Beispiel

Jahresabschluss

datum
15.01.2001

Kosten

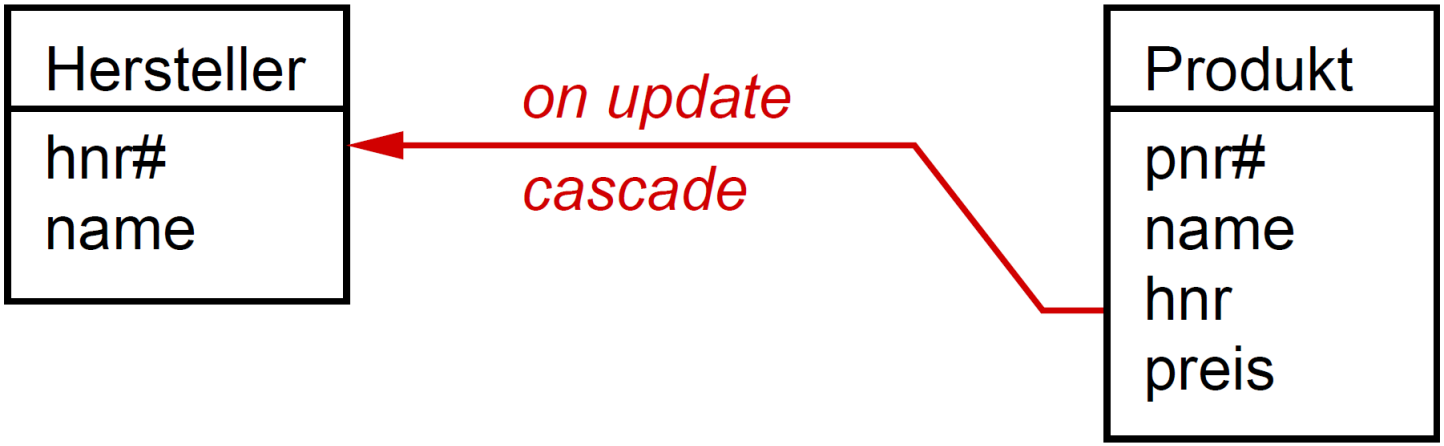
lnr#	artnr	kst	netto	datum
1	K001	7020	50.12	01.12.2000
2	U003	7030	82.50	29.02.2002

- Verhinderung unzulässiges Kostendatum
  - Nach erfolgtem Jahresabschluss dürfen keine Kosten davor mehr angelegt oder geändert werden
  - Tabelle Jahresabschluss enthält letztes Abschlussdatum
  - Trigger auf Kosten überprüft Integritätsbedingung

- Siehe plsqldemo.zip im DBS Moodle-Kurs

Trigger für Foreign Key Constraints

- Auch Foreign Key Constraints können über Trigger realisiert werden
- Wieviele und was für Trigger sind z.B. für folgenden Foreign Key Constraint erforderlich?



- Tatsächlich realisiert Postgres Foreign Key Constraints intern mit Triggern

Trigger zur Berechnung redundanter Werte

- Tabelle lieferung enthält Redundanzen wegen

$$brutto = netto * \frac{100+mwst}{100}$$

lnr#	produkt	menge	netto	mwst	brutto	datum
001	Buch A	1	49.35	7.0	52.80	01.12.2002
002	Buch B	1	116.94	7.0	125.13	05.01.2003
003	Software	1	38.90	16.0	43.40	01.08.2002

- Update-Anomalie
  - Explizites Speichern berechneter Attribute sollte vermieden werden, gelegentlich aber aus Performancegründen unvermeidbar
  - Wenn `netto` und/oder `mwst` geändert wird, muss zwangsläufig auch `brutto` geändert werden
  - Konsistenthaltung kann mit Trigger automatisiert werden