# Datenbanksysteme

Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

| Voraussetzen | |
|---|---|
| **vorgaenger** | **nachfolger** |
| 5001 | 5041 |
| 5001 | 5043 |
| 5001 | 5049 |
| 5041 | 5216 |
| 5043 | 5052 |
| 5041 | 5052 |
| 5052 | 5259 |

| Vorlesungen | |
|---|---|
| **vorlnr** | **titel** |
| 5001 | Grundzuege |
| 5041 | Ethik |
| 5043 | Erkenntnistheorie |
| 5049 | Maeeutik |
| 4052 | Logik |
| 5052 | Wissenschaftstheorie |
| 5216 | Bioethik |
| 5259 | Der Wiener Kreis |
| 5022 | Glaube und Wissen |
| 4630 | Die 3 Kritiken |

**Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?**

**Wie kann dies in SQL berechnet werden?**

# Rekursion

**Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?**

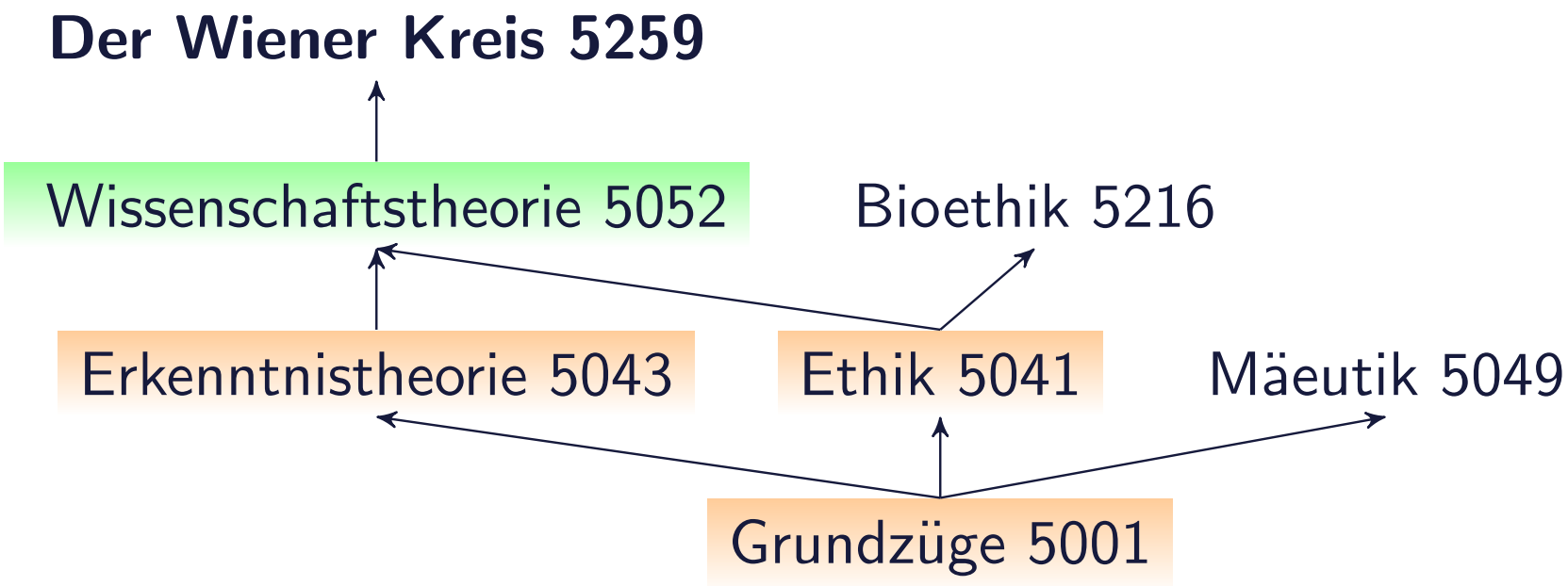
```
select Vorgaenger
from voraussetzen, Vorlesungen
where Nachfolger=VorlNr and
      Titel = 'Der Wiener Kreis';
```

# Rekursion

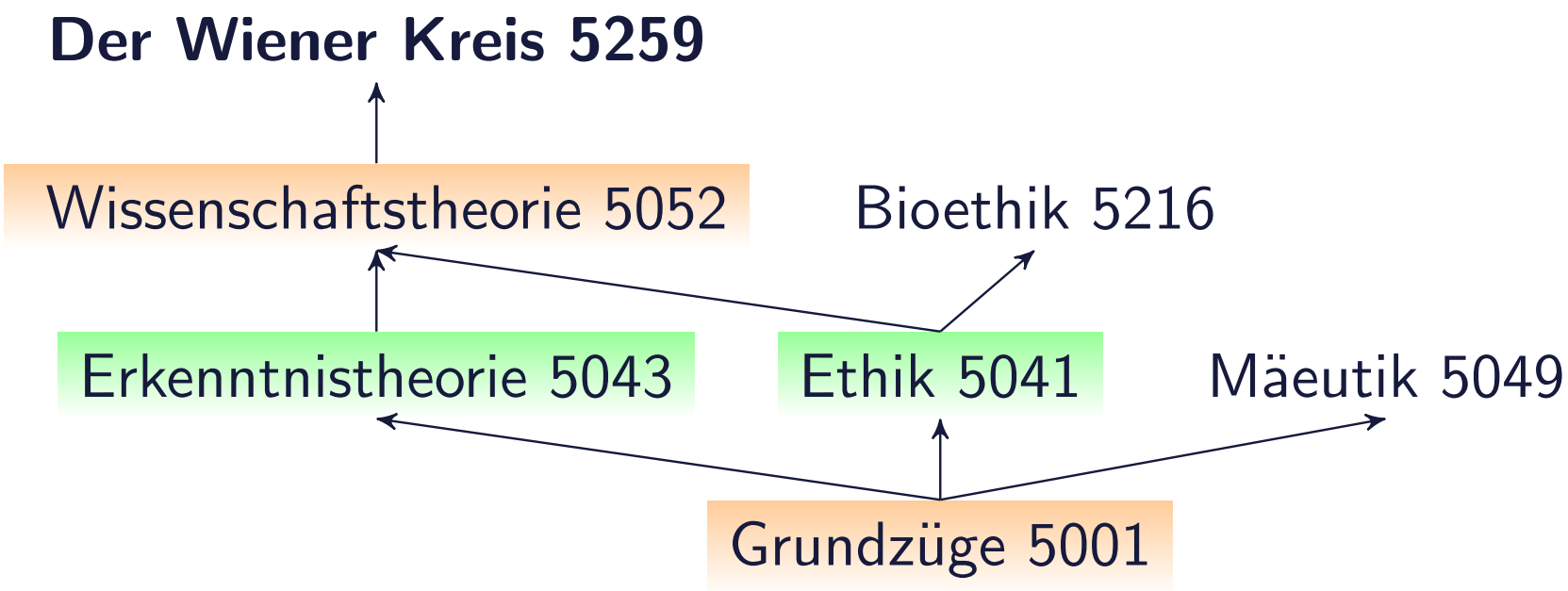
**Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?**

```
select Vorgaenger
from voraussetzen, Vorlesungen
where Nachfolger=VorlNr and
      Titel = 'Der Wiener Kreis';
```

Hier werden allerdings nur die direkten Vorgänger berechnet. Ergebnis: Vorlesung mit VorlNr=5052.


Der Wiener Kreis 5259
Wissenschaftstheorie 5052
Bioethik 5216
Erkenntnistheorie 5043
Ethik 5041
Mäeutik 5049
Grundzüge 5001

```
select v1.Vorgaenger
from voraussetzen v1, voraussetzen v2, Vorlesungen v
where v1.Nachfolger= v2.Vorgaenger and
v2.Nachfolger= v.VorlNr and
v.Titel='Der Wiener Kreis';
```



# Vorgänger der Tiefe n

```
select v1.Vorgaenger
from voraussetzen v1
     ...
     voraussetzen vn_minus_1
     voraussetzen vn,
     Vorlesungen v
where v1.Nachfolger= v2.Vorgaenger and
     ...
     vn_minus_1.Nachfolger= vn.Vorgaenger and
     vn.Nachfolger = v.VorlNr and
     v.Titel='Der Wiener Kreis';
```

**Wie kann man alle Vorgänger finden?**

Tiefe 1 union Tiefe 2 union Tiefe 3 union ...

# Transitive Hülle

Was hier sehr hilfreich wäre: **Berechnung der transitiven Hülle**.

$$trans_{A,B}(R) = \{(a,b)|\exists k \in \mathbb{N}(\exists \tau_1, \tau_2, ...\tau_k \in R($$

$$\tau_1.A = \tau_2.B \wedge$$

$$\tau_2.A = \tau_3.B \wedge$$

$$...$$

$$\tau_{k-1}.A = \tau_k.B \wedge$$

$$\tau_1.A = a \wedge$$

$$\tau_k.B = b))\}$$

**Enthält alle Paare, für die ein "Pfad" beliebiger Länge $k$ in $R$ existiert.**

# Values Statement

- Erzeugt konstante Tabelle
- **values** (1, 'eins'), (2, 'zwei'), (3, 'drei');

  ist äquivalent zu:

```
    select 1 as column1, 'eins' AS column2
union all
    select 2, 'zwei'
union all
    select 3, 'drei';
```

**Verwendung z.B. in Select Statement (auch in Joins) als normale Tabelle**

```
select * from
(values (1,'eins'), (2,'zwei') ) as table1 (nummer, wert);
```

# Vergleiche hierzu:

```
with mytable(mycolumn) as(
    values (1)
)
select mycolumn+1
from mytable;
```

# Rekursionen in SQL

Definieren eine rekursive "View", basierend auf Union (all) von Zwei Anfragen: Einer nicht-rekursiven Anfrage und einer rekursiven.

```
with recursive mytable(mycolumn) as (
        values(1)                          nicht rekursiver Teil
    union all                              union all
        select mycolumn+1
        from mytable                       rekursiver Teil:
        where mycolumn < 100               nur dieser darf mytable referenzieren
)
select sum(mycolumn)                       eigentlicher Aufruf
from mytable;
```

**Ergebnis: 5050**

Achtung: Darauf achten, dass die Rekursion terminiert!

# Auswertung von rekursiven Anfragen

## Schritt 1

- Evaluiere den nicht-rekursiven Teil. Für UNION (nicht aber für UNION ALL), entferne Duplikate. Alle verbliebenen Tupel werden in Ergebnis hinzugefügt und auch in eine temporäre Tabelle kopiert.

## Schritt 2: Solange temporäre Tabelle nicht leer ist wiederhole:

(a) Evaluiere den rekursiven Teil, wobei die Eigenreferenz durch die temporäre Tabelle ersetzt wird. Für UNION (aber nicht UNION ALL), entferne Duplikate und auch Duplikate zu vorherigen Ergebnissen. Füge verbliebene Tupel in Ergebnis hinzu und ebenso in eine temporäre Zwischenergebnis-Tabelle.

(b) Ersetzt Inhalt der temporären Tabelle mit dem Inhalt der Zwischenergebnis-Tabelle, leere die temporäre Zwischenergebnis-Tabelle.

# Rekursion: Anfrage für Voraussetzungen

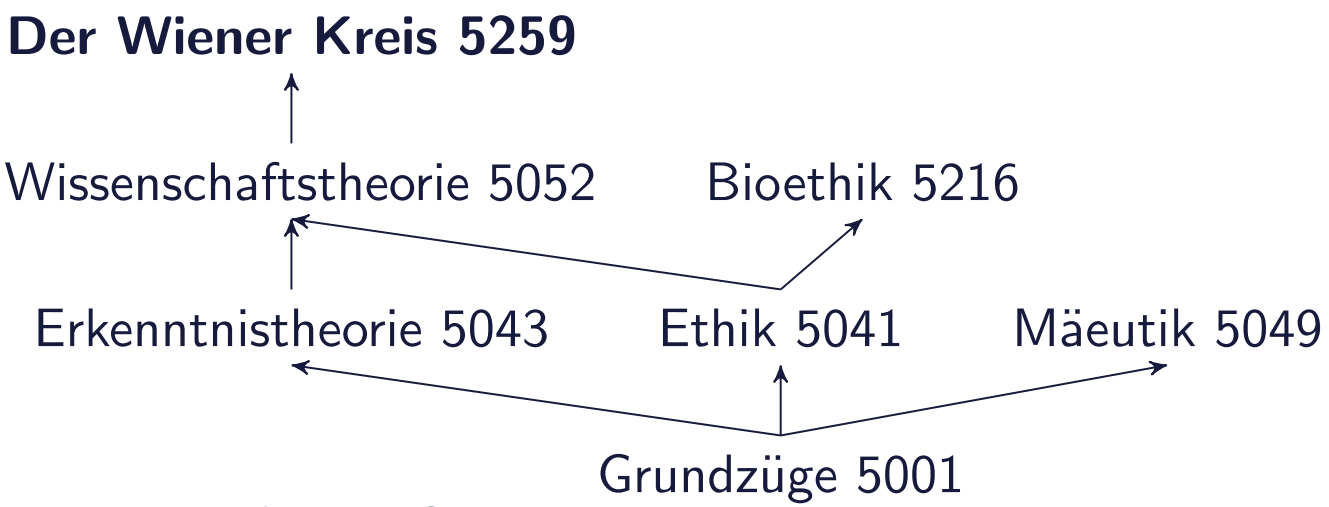
```
with recursive TransitivVorl (Vorg, Nachf)
as (
    select Vorgaenger, Nachfolger
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select *
from TransitivVorl
order by (vorg,nachf) asc;
```

# Rekursion: Anfrage für Voraussetzungen

```
with recursive TransitivVorl (Vorg, Nachf)
as (
    select Vorgaenger, Nachfolger
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select v2.titel
from TransitivVorl tv, vorlesungen v1, vorlesungen v2
where tv.nachf=v1.vorlnr and v1.titel='Der Wiener Kreis'
    and vorg=v2.vorlnr;
```

```
with recursive TransitivVorl (Vorg, Nachf, iteration)
as (
    select Vorgaenger, Nachfolger, 1 as iteration
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger, 1+ t.iteration
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select * from TransitivVorl order by iteration;
```

| | vorg integer | nachf integer | iteration integer |
|---|---|---|---|
| 1 | 5001 | 5041 | 1 |
| 2 | 5001 | 5043 | 1 |
| 3 | 5001 | 5049 | 1 |
| 4 | 5041 | 5216 | 1 |
| 5 | 5043 | 5052 | 1 |
| 6 | 5041 | 5052 | 1 |
| 7 | 5052 | 5259 | 1 |
| 8 | 5043 | 5259 | 2 |
| 9 | 5041 | 5259 | 2 |
| 10 | 5001 | 5052 | 2 |
| 11 | 5001 | 5216 | 2 |
| 12 | 5001 | 5259 | 3 |



# Nur zur Info: User-defined Function (UDF), die für eine bestimmte Vorlesung alle Vorgänger berechnet. UDFs betrachten wir später noch.

```
CREATE OR REPLACE FUNCTION alleVorgaenger(id integer)
RETURNS TABLE (VorlNr integer) AS $$
BEGIN
     -- temporaere Tabellen
    CREATE TEMP TABLE IF NOT EXISTS vorgaengerTabelle(VorlNr integer);
    CREATE TEMP TABLE IF NOT EXISTS neu_vorgaengerTabelle(VorlNr integer);
    CREATE TEMP TABLE IF NOT EXISTS temp_Tabelle(VorlNr integer);
    DELETE FROM vorgaengerTabelle;
    DELETE FROM temp_Tabelle;
    DELETE FROM neu_vorgaengerTabelle;

    --los geht es mit Suche der direkten Vorgaenger
    INSERT INTO neu_vorgaengerTabelle (select v.vorgaenger from voraussetzen v where v.nachfolger = id);
    LOOP
        INSERT INTO vorgaengerTabelle (select r.vorlnr from neu_vorgaengerTabelle r);
        INSERT INTO temp_Tabelle   --suche neue Vorgaenger
            (SELECT v.vorgaenger
            FROM neu_vorgaengerTabelle r, voraussetzen v
            WHERE v.nachfolger = r.vorlnr
            ) --aber nur Neue
            EXCEPT (SELECT r.vorlnr FROM vorgaengerTabelle r);
        DELETE from neu_vorgaengerTabelle;
        INSERT into neu_vorgaengerTabelle (select * from temp_Tabelle);
        DELETE FROM temp_Tabelle;
        IF NOT EXISTS (SELECT * FROM neu_vorgaengerTabelle) THEN
            EXIT; --exit aus Schleife, falls keine weiteren Vorgaenger gefunden
        END IF;
    END LOOP;
    RETURN QUERY SELECT  r.vorlnr FROM vorgaengerTabelle r;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM alleVorgaenger(5259)
```

# Rekursion in SQL

- Iteration wird so lange ausgeführt bis keine neuen Tupel hinzugefügt werden.
- Also ein Fixpunkt erreicht ist.

## Voraussetzungen

- Die rekursive Anfrage muss **monoton** sein, d.h. ausgeführt auf einer Instanz $V_1$ der rekursiven View muss Ergebnis eine Obermenge von den Ergebnissen auf Instanz $V_2$ sein, falls $V_1$ eine Obermenge von $V_2$ ist.
- D.h. wenn mehr Tupel zur View hinzugefügt werden muss die rekursive Anfrage mindestens die gleichen Tupel wie zuvor liefern.
- z.B. sollte die rekursive Anfrage also kein NOT EXISTS enthalten.

# Übersicht der (kommenden) Vorlesung(en)

- Embedded SQL (in Java und C++)
- Stored Procedures und User-Defined Functions
- Database Triggers

**Danach:**

- Transaktionsverwaltung (Recovery, Mehrbenutzersynchronisation)
- MapReduce und NoSQL
- ...

# Wiederholung: JDBC: Connect und einfache Anfrage

```
1  //registriere geeigneten Treiber (hier fuer
       Postgresql)
2  Class.forName("org.postgresql.Driver");
3  //erzeuge Verbindung zur Datenbank
4  Connection conn = DriverManager.getConnection(
5       "jdbc:postgresql://localhost/university",
6       "username", "password");
7
8  //erzeuge ein einfaches Statement Objekt
9  Statement stmt = conn.createStatement();
10
11 //mit execute Query koennen nun darauf Anfragen
       ausgefuehrt werden
12 //Ergebnisse in Form eines ResultSet Objekts
13 ResultSet rset = stmt.executeQuery("SELECT p.
       persnr from professoren p");
```

# Wiederholung: JDBC: Connect und einfache Anfrage

```
14 //dieses besitzt Metadaten
15 ResultSetMetaData metadata = rset.getMetaData();
16
17 //welche Attribute (Spalten) besitzen die
     Ergebnis-Tupel?
18 int column_count = metadata.getColumnCount();
19
20 for (int index=1; index<=column_count; index++)
     {
21       System.out.println("Spalte "+index+"
     heisst " +
22                metadata.getColumnName(index));
23 }
24 //iteriere nun ueber Ergebnisse
25 while (rset.next()) {
26       System.out.println(rset.getString(1));
27 }
```

# Call-level-Interface (CLI) vs. Embedded SQL

- Unter Verwendung einer Bibliothek werden aus dem Anwendungsprogramm (Wirtssprache) Funktionen aufgerufen, die mit dem DBMS kommunizieren.
- **JDBC ist ein Beispiel für ein CLI**
- **Im embedded SQL werden hingegen SQL Anweisungen direkt in der Wirtssprache benutzt**.
- (Dennoch werden diese letztendlich durch Aufrufe von DBMS-spezifischen Bibliotheken realisiert)

# Embedded SQL (ESQL)

## Idee

- Benutze SQL-Anweisungen direkt im Programmcode
- Syntax in Java:

```
#sql { <sql-statement> };
```

- Syntax in C oder C++

```
EXEC SQL <sql-statement>;
```

Zum Beispiel:

```
1  EXEC SQL
2  SELECT vorname, nachname
3  INTO :vorname, :nachname
4  FROM mitarbeitertabelle
5  WHERE pnr = :pnr;
```

# SQLJ: Embedded SQL für Java

- Einbettung von SQL in Java
- Anweisungen der Form

```
#sql { <sql-statement> };
```

Zum Beispiel:

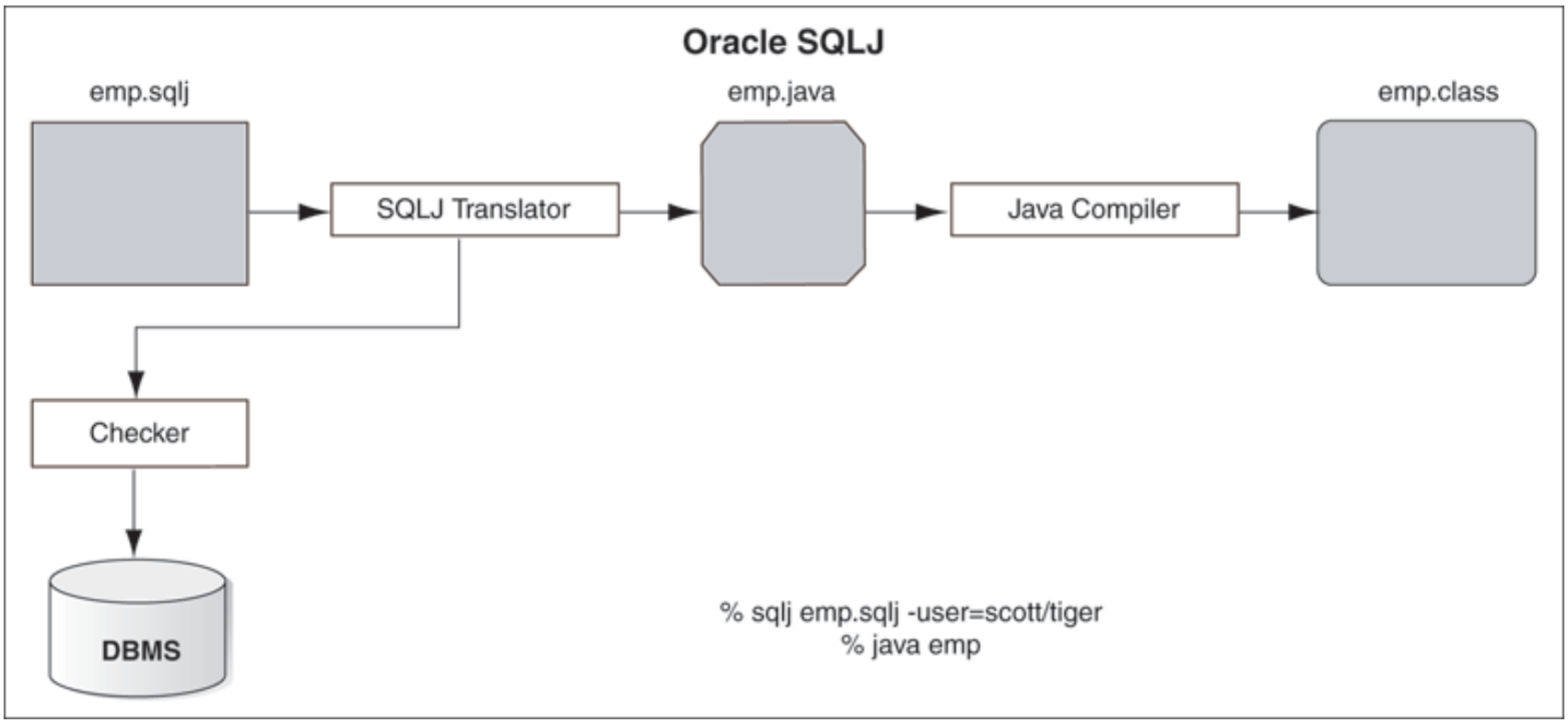
```
#sql {INSERT INTO emp (ename, sal)
      VALUES ('Joe', 43000) };
```

# SQLJ: Embedded SQL für Java

## Idee

- SQL Anweisungen werden direkt im Java Code benutzt (embedded)
- Ein Precompiler übersetzt diesen gemischten Code (in `*.sqlj` Dateien) in normalen Java Code (in `*.java` Dateien).
- Java Code benutzt dann JDBC oder Implementierung ähnlich zum JDBC Konzept.

# SQLJ: Schritte der Übersetzung




Check gegen DBMS ist optional.

Quelle:

`https://docs.oracle.com/cd/B28359_01/java.111/b31227/overview.htm`

# SQLJ: Vor- und Nachteile im Vergleich zu JDBC

## Vorteile

- Einfacher (kompakter) Code
- Verwendung der gleichen Variablen in SQL und in Java

## Nachteile

- Erfordert extra Übersetzung in "normales" Java.

## Umsetzung/Unterstützung

- Wird von Oracle angeboten (im eigenen DBMS)
- Sonst kaum (nicht) anzutreffen

# Beispiel

https://docs.oracle.com/cd/B28359_01/java.111/b31227/overview.htm

```
1  import java.sql.*;
2
3  /**   This is what you have to do in SQLJ   **/
4  public class SimpleDemoSQLJ
5  {
6    //TO DO: make a main that calls this
7    public Address getEmployeeAddress(int empno)
8      throws SQLException
9    {
10     Address addr;
11     #sql { SELECT office_addr INTO :addr FROM
       employees
12           WHERE empnumber = :empno };
13     return addr;
14   }
```

# Beispiel

```
15
16    public Address updateAddress(Address addr)
17      throws SQLException
18    {
19      #sql addr = { VALUES(UPDATE_ADDRESS(:addr))
        };
20      return addr;
21    }
22  }
```

- Vergleichbarer Code in JDBC ist um einiges länger
- Siehe obige URL (Oracle)

# Embedded SQL in C/C++

- Weit verbreitet, wird von vielen DBMS unterstützt
- Postgresql: ECPG compiler
- Oracle: Pro*C/C++ compiler

Embedded SQL ist standardisiert. Dies gilt allerdings nicht für Spracherweiterungen wie Oracles **PL/SQL** oder Postgresqls **PL/pgSQL**.

Microsoft hat mit LINQ (Language Integrated Query) einen zu embedded SQL ähnlichen Ansatz für das .NET Framework entwickelt.

# Beispiel

```
1  #include <stdio.h>
2  EXEC SQL BEGIN DECLARE SECTION;
3      int matrnr;
4      char name[1024];
5      int matrnrBound;
6  EXEC SQL END DECLARE SECTION;
7
8  int main()
9  {  EXEC SQL CONNECT TO unix:postgresql://
       localhost/university;
10
11   //select for single result items, otherwise
       use cursors
12   EXEC SQL SELECT matrnr INTO :matrnr from
       studenten where name = 'Fichte';
13   printf("matrnr=%s\n", matrnr);
```

```
14  //nun  Anfrage mit mehreren Ergebnissen
15  matrnrBound = 27000;
16  EXEC SQL DECLARE mycursor CURSOR FOR
17       SELECT matrnr, name
18       FROM studenten
19       WHERE matrnr < :matrnrBound
20       ORDER BY semester;
21  EXEC SQL OPEN mycursor;
22  EXEC SQL WHENEVER NOT FOUND DO BREAK;
23  while(1) {
24     EXEC SQL FETCH mycursor INTO :matrnr, :name;
25     printf("%i\t%s\n", matrnr, name);
26  }
27  EXEC SQL CLOSE mycursor;
28  EXEC SQL COMMIT;
29  EXEC SQL DISCONNECT ALL;
30  return 0;
31 }
```

# Embedded SQL in C/C++

## ECPG

- `http://www.postgresql.org/docs/9.3/interactive/ecpg-commands.html`

# Übersetzen von embedded SQL Code

Das Tool ecpg ist der Precompiler für Postgresqls embedded C.

- Eingabe ist eine Quellcode-Code in C mit eingebetteten SQL Befehlen.
- Ausgabe ist C-Code (Datei), der mittels der ECPG Bibliothek in der Lage ist mit der Postgresql Datenbank zu kommunizieren.

Übersetzung von embedded SQL in reines C:

```
ecpg test.c -o test_parsed.c
```

Kombilieren des C-Codes:

```
gcc test_parsed.c -o test -I /usr/include/postgresql/ -lecpg
```

# Übersetzter ECPG Code

Nur zur Veranschaulichung!

```
1  /* Processed by ecpg (4.8.0) */
2  /* These include files are added by the preprocessor */
3  #include <ecpglib.h>
4  #include <ecpgerrno.h>
5  #include <sqlca.h>
6  /* End of automatic include section */
7
8  #line 1 "test.c"
9
10 #include <stdio.h>
11
12 /* exec sql begin declare section */
13
14 #line 6 "test.c"
15   int matrnr ;
16
17 #line 7 "test.c"
18   char name [ 1024 ] ;
```

```
19 #line 8 "test.c"
20  int matrnrBound ;
21 /* exec sql end declare section */
22 #line 9 "test.c"
23
24 int main()
25 {
26     { ECPGconnect(__LINE__, 0, "unix:postgresql://
    localhost/university" , NULL, NULL , NULL, 0); }
27 #line 14 "test.c"
28
29     // as con1 USER smichel;
30     //EXEC SQL CONNECT TO unix:postgresql://localhost/
    university AS myconnection USER smichel;
31
32     //select for single result items, otherwise use
    cursors
33     { ECPGdo(__LINE__, 0, 1, NULL, 0, ECPGst_normal, "
    select matrnr from studenten where name = 'Fichte'",
    ECPGt_EOIT,
34     ECPGt_int,&(matrnr),(long)1,(long)1,sizeof(int),
35     ECPGt_NO_INDICATOR, NULL, 0L, 0L, 0L, ECPGt_EORT);}
```

```
36 #line 19 "test.c"
37      printf("matrnr=%i\n", matrnr);
38
39      //nun eine Anfrage, die mehrere Ergebnisse zurueck
        liefert
40
41      matrnrBound = 27000;
42
43      /* declare mycursor cursor for select matrnr , name
        from studenten where matrnr < $1  order by semester
        */
44 #line 31 "test.c"
45
46
47      { ECPGdo(__LINE__, 0, 1, NULL, 0, ECPGst_normal, "
        declare mycursor cursor for select matrnr , name from
         studenten where matrnr < $1  order by semester",
48      ECPGt_int,&(matrnrBound),(long)1,(long)1,sizeof(int),
49      ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_EORT);}
```

```
50 #line 33 "test.c"
51      /* exec sql whenever not found  break ; */
52 #line 34 "test.c"
53
54      while(1) {
55          { ECPGdo(__LINE__, 0, 1, NULL, 0, ECPGst_normal, "
       fetch mycursor", ECPGt_EOIT,
56      ECPGt_int,&(matrnr),(long)1,(long)1,sizeof(int),
57      ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L,
58      ECPGt_char,(name),(long)1024,(long)1,(1024)*sizeof(
       char),
59      ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
60 #line 36 "test.c"
61
62  if (sqlca.sqlcode == ECPG_NOT_FOUND) break;}
63 #line 36 "test.c"
64
65          printf("%i\t%s\n", matrnr, name);
66      }
```

```
67      { ECPGdo(__LINE__, 0, 1, NULL, 0, ECPGst_normal, "
    close mycursor", ECPGt_EOIT, ECPGt_EORT);}
68 #line 40 "test.c"
69
70      { ECPGtrans(__LINE__, NULL, "commit");}
71 #line 41 "test.c"
72
73      { ECPGdisconnect(__LINE__, "ALL");}
74 #line 42 "test.c"
75
76      return 0;
77 }
```

# Aufbau einer Verbindung zur DB

```
#include <stdio.h>

int main()
{
    EXEC SQL CONNECT TO
             unix:postgresql://localhost/university;
    //...
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

# Aufbau einer Verbindung zur DB (2)

- Es können auch mehrere Verbindungen aufgebaut werden und via Namen benutzt werden:

```
EXEC SQL CONNECT TO
        unix:postgresql://localhost/university;
        AS conn1;
```

- Weitere Parameter wie Login, Passwort, Port sind natürlich ebenfalls möglich

# Host-Variablen

Die sogennanten Host-Variablen sind Variablen, die gemeinsam vom Programmcode und SQL Anweisungen benutzt werden können.
Der Name der C/C++ Variablen wird in SQL mit einem Doppelpunkt als Präfix benutzt. Z.B.

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo
    ', :v2);
```

## Deklaration

Host-Variablen müssen speziell deklariert werden:

```
EXEC SQL BEGIN DECLARE SECTION;
    int x=4;
    char foo[16], bar[16];
EXEC SQL END DECLARE SECTION;
```

http://www.postgresql.org/docs/9.3/interactive/ecpg-variables.html

# Transaktionen

- Abschließen einer Transaktion

  ```
  EXEC SQL COMMIT
  ```

- Rollback der aktuellen Transaktion

  ```
  EXEC SQL ROLLBACK
  ```

- Ein- bzw. Abschalten des automatischen Commits

  ```
  EXEC SQL SET AUTOCOMMIT TO ON
  EXEC SQL SET AUTOCOMMIT TO OFF
  ```

# Arbeiten mit Cursorn

Gibt eine Anweisung mehrere Zeilen zurück müssen Cursor benutzt werden. In JDBC-Terminologie ist ein ResultSet ein Cursor.

```
1 ....
2 matrnrBound = 27000;
3
4
5 EXEC SQL DECLARE mycursor CURSOR FOR
6       SELECT matrnr, name
7       FROM studenten
8       WHERE matrnr < :matrnrBound
9       ORDER BY semester;
```

- Die Host-Variable matrnBound wird hier direkt in der SQL-Anweisung benutzt.
- Man könnte auch eine parametrisierte SQL Anweisung benutzten. Wie wurde diese Klasse in JDBC genannt?

# Arbeiten mit Cursorn (2)

- Der Cursor muss nun nur noch geöffnet werden
- Dann kann via FETCH auf die einzelnen Tupel bzw. Spalten zugegriffen werden.

```
1  //cursor wird geoeffnet
2  EXEC SQL OPEN mycursor;
3  //was soll geschehen wenn keine Ergebnisse
       geliefert werden?
4  EXEC SQL WHENEVER NOT FOUND DO BREAK;
5
6  //solange kein BREAK aufgerufen wird laufe ueber
       Zeilen
7  while(1) {
8      EXEC SQL FETCH mycursor INTO :matrnr, :name;
9      printf("%i\t%s\n", matrnr, name);
10 }
11
```

# Prepared-Statements

Ähnlich zu JDBC können wir auch in embedded SQL Prepared-Statements benutzen.

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname
    FROM pg_database WHERE oid = ?";
```

Bei der (bzw. vor der) Ausführung müssen dann die freien Parameter gesetzt werden. Zudem wird erst jetzt angegeben in welchen Host-Variablen die Attribute des Ergebnis-Tupels gespeichert werden soll.

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname
   USING 1;
```

Wenn das Prepared-Statement nicht mehr gebraucht wird:

```
EXEC SQL DEALLOCATE PREPARE name;
```

# Prepared-Statements und Cursor

Hier wird ein Prepared-Statement erzeugt und dann ein Cursor darüber definiert:

```
1  EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname
       FROM pg_database WHERE oid > ?";
2  EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;
3
4  // wenn Ende erreicht ist, mittels BREAK aus
       While-Schleife aussteigen
5  EXEC SQL WHENEVER NOT FOUND DO BREAK;
6
7  EXEC SQL OPEN foo_bar USING 100;
8  ...
9  while (1)
10 {
11     EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid
       , :dbname;
```

# Allgemein: Prepared-Statements, Static vs. Dynamic SQL

## Übersetzung von Anfragen im DBMS

- Tritt eine Anfrage zum ersten Mal auf, muss sie "übersetzt" werden (compiled).
- D.h. Syntaxprüfung, Prüfung von Rechten, Anfrageoptimierung, Code-Generierung, etc.

## Wiederverwendung von kompilierten Anfragen

- Sind Anfragen parametrisiert, wie im Falle von Prepared-Statements, so kann das DBMS den bereits erzeugten Plan wiederverwenden.
- Im Vergleich zu nicht parametrisierten Statements, fallen hier die Kosten für die Übersetzung nur ein Mal an.

# Allgemein: Prepared-Statements, Static vs. Dynamic SQL

## Wiederverwendung von kompilierten Anfragen

- DBMS prüft bei Eintreffen einer Anfrage ob Plan bereits existiert.
- Dazu werden Pläne in Cache gehalten ($\rightarrow$Ersetzungsstrategien)

## Neu-Kompilierung bestehender Pläne

Falls sich Eigenschaften der DB ändern die essentiell für Plangenerierung sind. Zum Beispiel:

- Indexe werden hinzugefügt oder gelöscht.
- Sehr viele Änderungen an Daten der relevanten Tabellen
- Explizite Anweisungen neu zu kompilieren, bzw.
- neue Statistiken verfügbar

# CLIs für Postgresql

## Für C++: libpqxx

- http://pqxx.org/

## Für Ruby: pg

- https://rubygems.org/gems/pg

```
require 'pg'
conn = PG::Connection.open(:host => 'localhost',
         :dbname => 'university', :user => 'username',
         :password => 'my password')
res = conn.exec("select name from studenten")
res.each do |row|
    puts row['name']
end
```

# Stored Procedures / User-Defined Functions

# Stored Procedures/UDFs

Bislang: Kommunikation mit DBMS via Anwendungsprogrammen: Einzelne Statements, Verarbeitung in Wirtssprache.
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 Performance
- Code kann wiederverwendet werden (zwischen Anwendungen)

## Nachteile

- Etwas aufwendiger zu erstellen.
- Debugging schwieriger.

# SQL vs. SQL/PSM vs. PL/SQL bzw. PL/pgSQL

## SQL

- Standard Query Language für Datenbanksysteme. Deklarativ.
- SQL Anweisungen können via Anwendungsprogrammierung (JDBC oder Embedded SQL) an DB geschickt werden.

## PSM

- Persistent, Stored Modules (PSM), bzw. Sprache diese zu realisieren.
- Im SQL:2003 Standard definiert. Erlaubt es prozeduralen Code direkt innerhalb der DB zu schreiben.

## PL/SQL bzw. PL/pgSQL

- Procedural Language/(PostgreSQL) Structured Query Language
- Prozedurale Sprache, benutzt in Oracle bzw. Postgresql
- Inspiriert von bzw. implementiert PSM.
- PL/SQL bzw. PL/pgSQL erlauben diese Anwendungslogik als Prozedur innerhalb des DBMS zu definieren.

# Vorteile von Stored Procedures / User Defined Functions

- **Ausführungspläne können vor-übersetzt werden**, sind **wiederverwendbar**
- **Anzahl der Zugriffe** des Anwendungsprogramms auf das DBMS werden reduziert, ebenso wie Menge an Daten, die zwischen Anwendung und DBMS hin und her geschickt wird.
- Prozeduren sind als **gemeinsamer Code** für verschiedene Anwendungsprogramme nutzbar
- Es wird ein **höherer Isolationsgrad** der Anwendung von dem DBMS erreicht.

# Beispiel

```
CREATE  FUNCTION wieVieleVL (_matrnr int)
                                    RETURNS int AS $$
DECLARE
    qty int;
BEGIN
    SELECT COUNT(*) INTO qty
    FROM hoeren h
    WHERE h.matrnr = _matrnr;

    RETURN qty;
END;
$$ LANGUAGE plpgsql;
```

```
select *
from wieVieleVL(28106);
```

Liefert Ergebnis: 4

# Unterschied Stored Procedures und UDFs

## Generell

- UDF = user-defined function
- Stored procedure muss explizit mit CALL aufgerufen werden (`SQL EXEC CALL name`))
- UDF kann direkt in SQL ohne CALL benutzt werden

## In Postgresql

- In Postgres (9.3) gibt es allerdings keinen Unterschied zwischen stored procedures und UDFs
- UDF wird aufgerufen in SELECT statements, z.b.
  **select from** myFunction(44234234);

# UDFs in Postgresql

## Verschiedene Arten von UDFs

- Query language Funktionen (SQL)
- Procedural language Funktionen (PL/pgSQL, Perl, ...)
- Interne Funktionen
- C Funktionen
- PL/Java erlaubt auch die Nutzung von Java (`http://pgfoundry.org/projects/pljava/`)

`http://www.postgresql.org/docs/9.3/static/xfunc.html`

# Query Language (SQL) Funktionen

Mit Hilfe des Schlüsselworts **LANGUAGE** wird angegeben welche Sprache zur Definition dieser Funktion benutzt wurde, hier SQL.

- Diese Funktion hat den Rückgabewert `void`
- Sie ist definiert als einfaches SQL delete Statement und besitzt auch keine Eingabeparameter.

```
CREATE FUNCTION clean_emp() RETURNS
void AS $$
    DELETE FROM emp
    WHERE salary < 0;
$$ LANGUAGE SQL;
```

# Query Language Funktionen (2)

- Diese Funktion hat als Parameter ein Tupel der Relation **emp**, die neben Name des Mitarbeiters, dessen Gehalt (Salary), Alter und Raum (Als Point-Objekt) enthält.
- Der Rückgabewert ist Typ `numeric`

```
INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp)
 RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;
```

# Query Language Funktionen (3)

```
CREATE FUNCTION addtoroom (professoren)
RETURNS int AS $$
select $1.raum+1 ;
$$ LANGUAGE SQL
```

Anwendung/Aufruf:

**select name, addtoroom(professoren.*) from professoren;**

# Query Language Funktionen (4)

Hier wird eine Funktion definiert, die ein Dummy-Tupel für einen neuen Professor erzeugt (gemäß der Relation `professoren`):

```
CREATE FUNCTION neuerProf()
RETURNS professoren
AS $$
    SELECT 1 as persnr, text 'Unbekannt' AS name,
           text 'C3' as rang, 123 as raum;
$$ LANGUAGE SQL;
```

Anwendung zum Beispiel:
insert into professoren (select * from neuerProf());

# Query Language Funktionen (5)

Diese Funktion hat mehrere Eingaben und mehrere Ausgaben:

```
CREATE FUNCTION sum_n_product
       (x int, y int, OUT sum int, OUT product int)
AS $$ SELECT $1 + $2, $1 * $2
$$ LANGUAGE SQL;
```

Beispielaufruf:

```
 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

# Query Language Funktionen (6)

Ruckgabewerte: Einzelne Zeilen vs. Tabellen

```
CREATE FUNCTION alleProfs()
RETURNS professoren
as $$
select * from professoren;
$$ LANGUAGE SQL;
```

Beispielaufruf:

```
select * from alleProfs();
```

```
 persnr |   name    | rang | raum
--------+-----------+------+------
   2125 | Sokrates | C4   |  226
(1 row)
```

Was macht `select alleProfs();` ?

# 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)
```

# PL/pgSQL

Anstelle von SQL in den vorherigen Beispielen wird nun PL/pgSQL betrachtet.

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

# PL/pgSQL

Anstelle von SQL in den vorherigen Beispielen wird nun PL/pgSQL betrachtet.

```
CREATE FUNCTION sales_tax(subtotal real)
RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

# PL/pgSQL

```
CREATE FUNCTION
    concat_selected_fields(in_t sometablename)
RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

# PL/pgSQL

Funktion mit zwei Eingabeparametern und zwei Ausgabeparametern:

```
CREATE FUNCTION
            sum_n_product(x int, y int,
                  OUT sum int, OUT prod int)
AS $$
BEGIN
     sum := x + y;
     prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

# 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 Ergebnis zu berücksichtigen:
`http://www.postgresql.org/docs/9.3/static/plpgsql-statements.html`

# PL/pgSQL: Kontrollstrukturen

PL/pgSQL bietet die übliche Auswahl and Kontrollstrukturen wie IF-Statements und Schleifen (LOOP WHILE, FOR), EXIT (=break), CONTINUE

```
LOOP
    IF count > 0 THEN
        EXIT;
    END IF;
END LOOP;
```

# 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. aequivalent: 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.