Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Übersicht der (kommenden) Vorlesungen

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

Wiederholung: JDBC: Connect und einfache Anfrage

```
//registriere geeigneten Treiber (hier fuer Postgresql)
Class.forName("org.postgresql.Driver");
//erzeuge Verbindung zur Datenbank
Connection conn = DriverManager.getConnection(
      "jdbc:postgresql://localhost/university",
      "username", "password");

//erzeuge ein einfaches Statement Objekt
Statement stmt = conn.createStatement();

//mit execute Query koennen nun darauf Anfragen ausgefuehrt
    werden
//Ergebnisse in Form eines ResultSet Objekts
ResultSet rset = stmt.executeQuery("SELECT p.persnr from
    professoren p");
```

Wiederholung: JDBC: Connect und einfache Anfrage

```
//dieses besitzt Metadaten
ResultSetMetaData metadata = rset.getMetaData();

//welche Attribute (Spalten) besitzen die Ergebnis-Tupel?
int column_count = metadata.getColumnCount();

for (int index=1; index<=column_count; index++) {
      System.out.println("Spalte "+index+" heisst " +
            metadata.getColumnName(index));
}

//iteriere nun ueber Ergebnisse
while (rset.next()) {
      System.out.println(rset.getString(1));
}
```

Call-level-Interface (CLI)

- 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:

```
EXEC SQL
SELECT vorname, nachname
INTO :vorname, :nachname
FROM mitarbeitertabelle
WHERE pnr = :pnr;
```

SQLJ: Embedded SQL für Java

- Einbettung von SQL in Java
- Anweisungen der Form

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

Zum Beispiel:

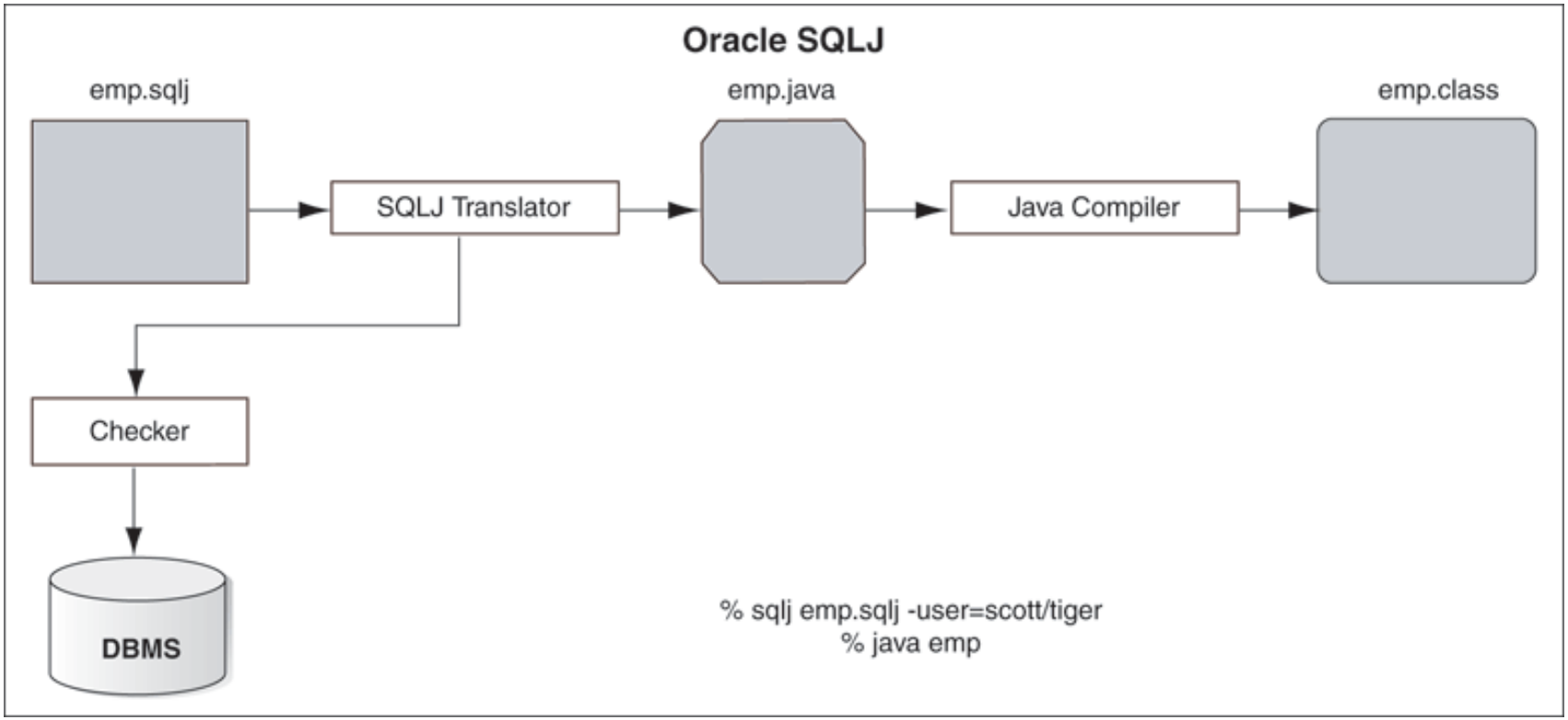
```
#sql {INSERT INTO emp (ename, sal)
      VALUES ('Joe', 43000) };
```

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).

SQLJ: Schritte der Übersetzung



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  /**
4    This is what you have to do in SQLJ
5    **/
6  public class SimpleDemoSQLJ
7  {
8    //TO DO: make a main that calls this
9
10   public Address getEmployeeAddress(int empno)
11     throws SQLException
12   {
13     Address addr;
14     #sql { SELECT office_addr INTO :addr FROM employees
15             WHERE empnumber = :empno };
16     return addr;
17   }
```

Beispiel

```
18
19    public Address updateAddress(Address addr)
20      throws SQLException
21    {
22      #sql addr = { VALUES(UPDATE_ADDRESS(:addr)) };
23      return addr;
24    }
25  }
```

- 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
3  EXEC SQL BEGIN DECLARE SECTION;
4      int matrnr;
5      char name[1024];
6      int matrnrBound;
7  EXEC SQL END DECLARE SECTION;
8
9  int main()
10 {
11   EXEC SQL CONNECT TO unix:postgresql://localhost/university;
12
13   //select for single result items, otherwise use cursors
14   EXEC SQL SELECT matrnr INTO :matrnr from studenten where
       name = 'Fichte';
15
16   printf("matrnr=%s\n", matrnr);
```

Beispiel (2)

```
17      //nun eine Anfrage, die mehrere Ergebnisse liefert
18
19      matrnrBound = 27000;
20
21      EXEC SQL DECLARE mycursor CURSOR FOR
22           SELECT matrnr, name
23           FROM studenten
24           WHERE matrnr < :matrnrBound
25           ORDER BY semester;
26
27      EXEC SQL OPEN mycursor;
28      EXEC SQL WHENEVER NOT FOUND DO BREAK;
29      while(1) {
30         EXEC SQL FETCH mycursor INTO :matrnr, :name;
31         printf("%i\t%s\n", matrnr, name);
32      }
33      EXEC SQL CLOSE mycursor;
34      EXEC SQL COMMIT;
35      EXEC SQL DISCONNECT ALL;
36      return 0;
37  }
```

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 ] ;
```

Übersetzter ECPG Code (2)

```
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);}
```

Übersetzter ECPG Code (3)

```
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);}
```

Übersetzter ECPG Code (4)

```
#line 33 "test.c"
     /* exec sql whenever not found  break ; */
#line 34 "test.c"

     while(1) {
        { ECPGdo(__LINE__, 0, 1, NULL, 0, ECPGst_normal, "
     fetch mycursor", ECPGt_EOIT,
     ECPGt_int,&(matrnr),(long)1,(long)1,sizeof(int),
     ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L,
     ECPGt_char,(name),(long)1024,(long)1,(1024)*sizeof(char),
     ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 36 "test.c"

 if (sqlca.sqlcode == ECPG_NOT_FOUND) break;}
#line 36 "test.c"

        printf("%i\t%s\n", matrnr, name);
     }
```

Übersetzter ECPG Code (5)

```
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.

```
....
matrnrBound = 27000;
```

```
EXEC SQL DECLARE mycursor CURSOR FOR
    SELECT matrnr, name
    FROM studenten
    WHERE matrnr < :matrnrBound
    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.

```
//cursor wird geoeffnet
EXEC SQL OPEN mycursor;
//was soll geschehen wenn keine Ergebnisse geliefert werden?
EXEC SQL WHENEVER NOT FOUND DO BREAK;

//solange kein BREAK aufgerufen wird laufe ueber Zeilen
while(1) {
   EXEC SQL FETCH mycursor INTO :matrnr, :name;
   printf("%i\t%s\n", matrnr, name);
}

//cursor wird geschlossen
EXEC SQL CLOSE mycursor;
```

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:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM
    pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

// wenn Ende erreicht ist, mittels BREAK aus While-Schleife
    aussteigen
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

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

Andere 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 Performanz
- 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.
- SQL Anweisungen können via Anwendungsprogrammierung (JDBC oder Embedded SQL) an DB geschickt werden.

PSM

- Persistent, Stored Modules (PSM)
- 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
- PL/SQL bzw. PL/pgSQL erlauben diese Anwendungslogik als Prozedur innerhalb des DBMS zu definieren.

Vorteile von Stored Procedures / User Defined Functions

- Ausführungspläne können vorübersetzt werden, sind wiederverwendbar
- Anzahl der Zugriffe des Anwendungsprogramms auf das DBMS werden reduziert
- Prozeduren sind als gemeinsamer Code für verschiedene Anwendungsprogramme nutzbar
- Es wird ein höherer Isolationsgrad der Anwendung von dem DBMS erreicht.

Beispiel

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

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

```
select *
from getstudent(26120);
```

Unterschied Stored Procedures und UDFs

Generel

- 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 Eregebnis zu berücksitigen:
`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;
```