8. Programmlogik und SQL

Vorlesung "Informationssysteme"

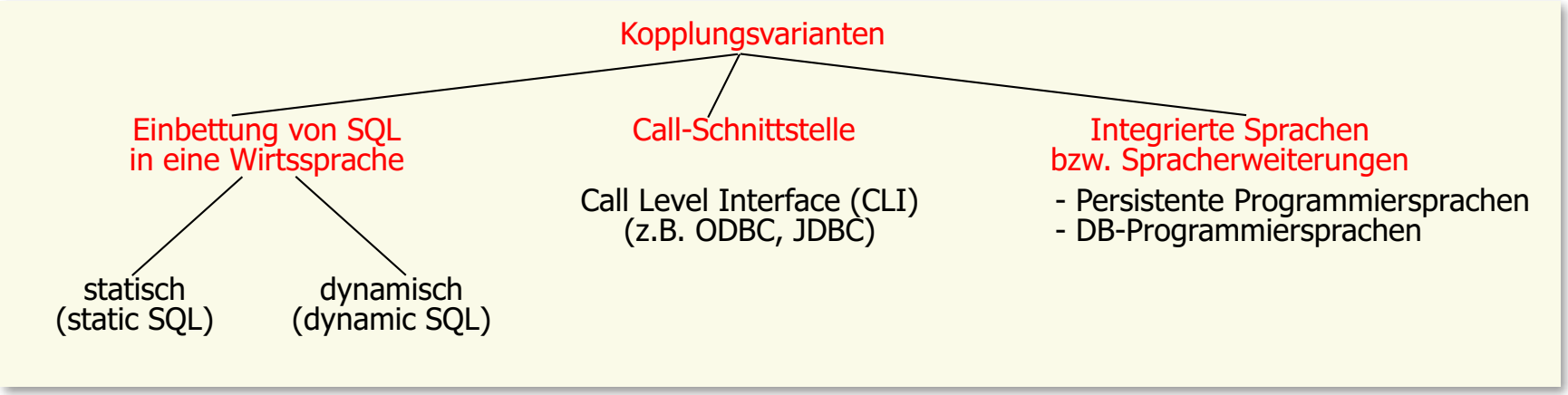
Sommersemester 2017

Technische Universität
Kaiserslautern

Gliederung

- Datenbankzugriff in Programmen
  - Kopplung mit einer Wirtssprache
    - Übersicht und Aufgaben
  - Eingebettetes statisches SQL
    - Cursor-Konzept
    - SQL-Programmiermodell
  - CLI und ODBC
  - DB-Zugriff aus Java-Programmen
    - DB-Zugriff via JDBC
    - SQLJ
- Benutzerdefinierte Prozeduren und Funktion in SQL

Kopplung mit einer Wirtssprache (*host language*)

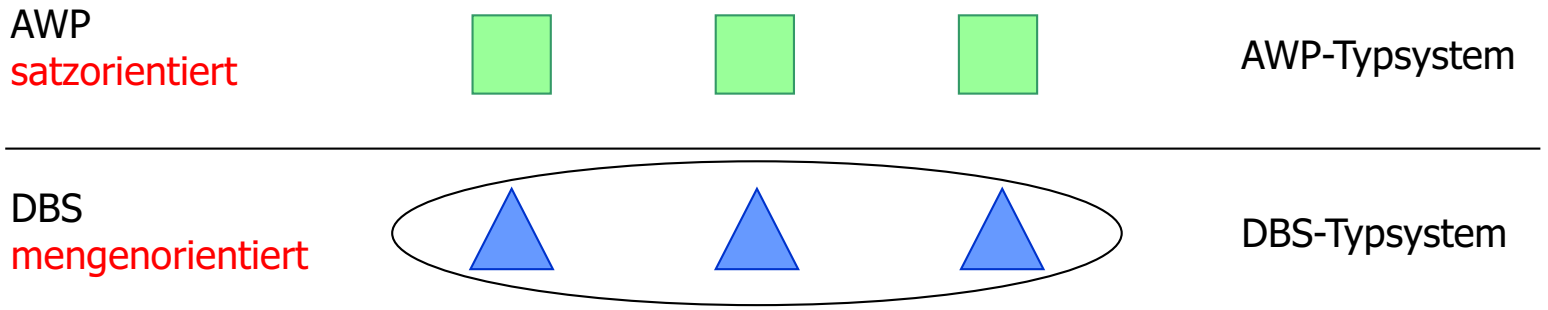


- Einbettung von SQL (Embedded SQL, ESQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - komfortablere Programmierung als mit CLI
- Statische Einbettung
  - Vorübersetzer (Precompiler) wandelt DB-Aufrufe in Prozeduraufrufe um
  - Nutzung der normalen PS-Übersetzer für umgebendes Programm
  - SQL-Anweisungen müssen zur Übersetzungszeit feststehen
  - im SQL-Standard unterstützte PS: C, COBOL, FORTRAN, Ada, PL1, Pascal, MUMPS, Java
- Dynamische Einbettung
  - Konstruktion von SQL-Anweisungen zur Laufzeit
- Call-Schnittstelle (prozedurale Schnittstelle, CLI)
  - DB-Funktionen werden durch Bibliothek von Prozeduren/Funktionen/Klassen realisiert
  - Anwendung enthält lediglich API-Aufrufe

Kopplung mit einer Wirtssprache (2)

- Wünschenswert sind Mehrsprachenfähigkeit und deskriptive DB-Operationen (mengenorientierter Zugriff)
- Relationale AP-Schnittstellen (API) bieten diese Eigenschaften, erfordern jedoch Maßnahmen zur Überwindung der sog. Fehlanpassung (impedance mismatch)
- Kernprobleme der API bei konventionellen Programmiersprachen
  - Konversion und Übergabe von Werten
  - Übergabe aktueller Werte von Wirtssprachenvariablen (Parametrisierung von DB-Operationen)
  - DB-Operationen sind i. allg. mengenorientiert: Wie und in welcher Reihenfolge werden Zeilen/Sätze dem AP zur Verfügung gestellt?

➥ Cursor-Konzept



Eingebettetes statisches SQL

- Anbindung einer SQL-Anweisung an die Wirtssprachen-Umgebung
  - SQL-Anweisungen werden durch "exec sql" eingeleitet und durch spezielles Symbol (z.B. ";") beendet, um dem Compiler eine Unterscheidung von anderen Anweisungen zu ermöglichen
  - Verwendung von AP-Variablen (Host-Variablen) in SQL-Anweisungen verlangt Deklaration innerhalb eines declare section-Blocks sowie Angabe des Präfix ":" innerhalb von SQL-Anweisungen
  - Kommunikationsbereich SQLCA (Rückgabe von Statusanzeigern u.ä.)
  - Übergabe der Werte einer Zeile mit Hilfe der INTO-Klausel
    - INTO target-commalist (Variablenliste des Wirtsprogramms)
    - Anpassung der Datentypen (Konversion)
  - Aufbau/Abbau einer Verbindung zu einem DBS: connect/disconnect

Eingebettetes statisches SQL – Beispiel für C

```
exec sql include sqla; /* SQL Communication Area */
main ()
{
exec sql begin declare section;
    char X[3] ;
    int   GSum;
exec sql end declare section;
exec sql connect to dbname;
exec sql insert into Pers (Pnr, Name) values (4711, 'Ernie');
exec sql insert into Pers (Pnr, Name) values (4712, 'Bert');
printf ("Anr?") ; scanf ("%s", X);
exec sql select sum (Gehalt) into :GSum from Pers where Anr = :X;
/* Es wird nur ein Ergebnissatz zurückgeliefert */
printf ("Gehaltssumme: %d\n", GSum)
exec sql commit work;
exec sql disconnect;
}
```

Cursor-Konzept

- Cursor-Konzept zur satzweisen Abarbeitung von Ergebnismengen
  - Cursor ist ein Iterator, der einer Anfrage zugeordnet wird und mit dessen Hilfe die Zeilen der Ergebnismenge einzeln (*one tuple at a time*) im Programm bereitgestellt werden
- Cursor-Deklaration

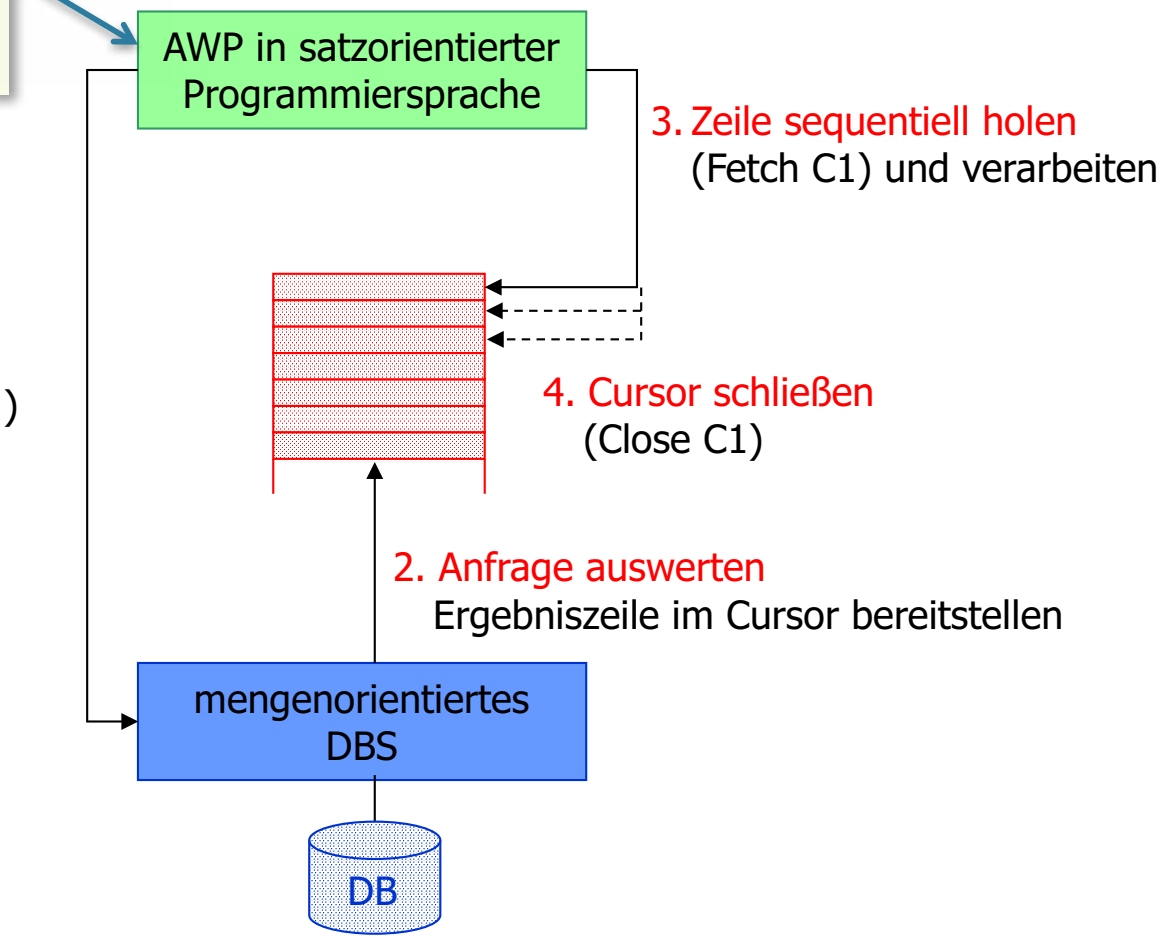
```
DECLARE  C1 CURSOR FOR
      SELECT Name, Gehalt, Anr
      FROM Pers WHERE Anr = 'K55'
      ORDER BY Name;
```

- Operationen auf einen Cursor C1
  - OPEN C1
  - FETCH C1 INTO :Var_1, :Var_2, …, :Var_n
  - CLOSE C1

Veranschaulichung der Cursor-Schnittstelle

Cursor-Deklation in AWP:

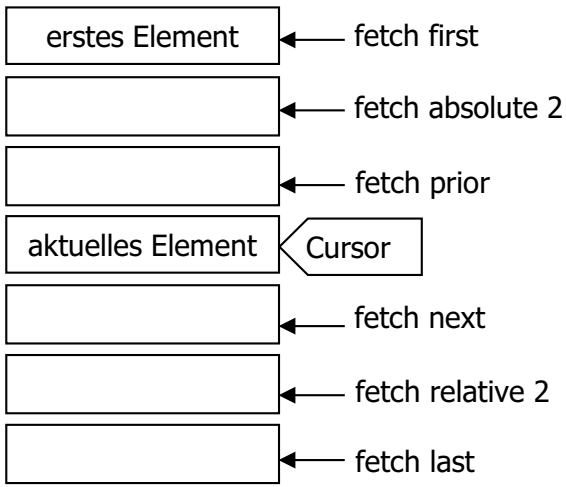
```
DECLARE C1 CURSOR FOR
        SELECT *
        FROM PERS
        WHERE ANR > '30'
        ORDER BY PNAME ASC
```



UPDATEBLE/HOLDABLE/SCROLLABLE Cursor

- Aktualisierung mit Bezugnahme auf eine Position
  - Wenn die Zeilen, die ein Cursor verwaltet (active set), eindeutig Zeilen einer Tabelle entsprechen, können sie über Bezugnahme durch den Cursor geändert/gelöscht werden.
  - Syntax: UPDATE/DELETE ... WHERE CURRENT OF C1;

  ➔ Änderbarkeit analog zu änderbaren Sichten definiert!
- Cursor kann explizit als "read only" bzw "updatable" deklariert werden (→ Optimierungspotential für DBMS)
  - Beispiel: DECLARE C1 CURSOR FOR SELECT ... **FOR READ ONLY**
- Holdable Cursor
  - Cursor werden spätestens bei Ende der Transaktion geschlossen
  - dies läßt sich mit der Option WITH HOLD verhindern!
    - Beispiel: DECLARE C1 CURSOR **WITH HOLD** FOR SELECT ...
- Scrollable Cursor
  - SCROLL-Option erlaubt flexible Positionierung
  - NEXT, PRIOR, FIRST, LAST, ABSOLUTE n, RELATIVE n
    - Beispiel: DECLARE C1 **SCROLL** CURSOR FOR SELECT ...
  - NO SCROLL (Default-Einstellung): nur NEXT möglich



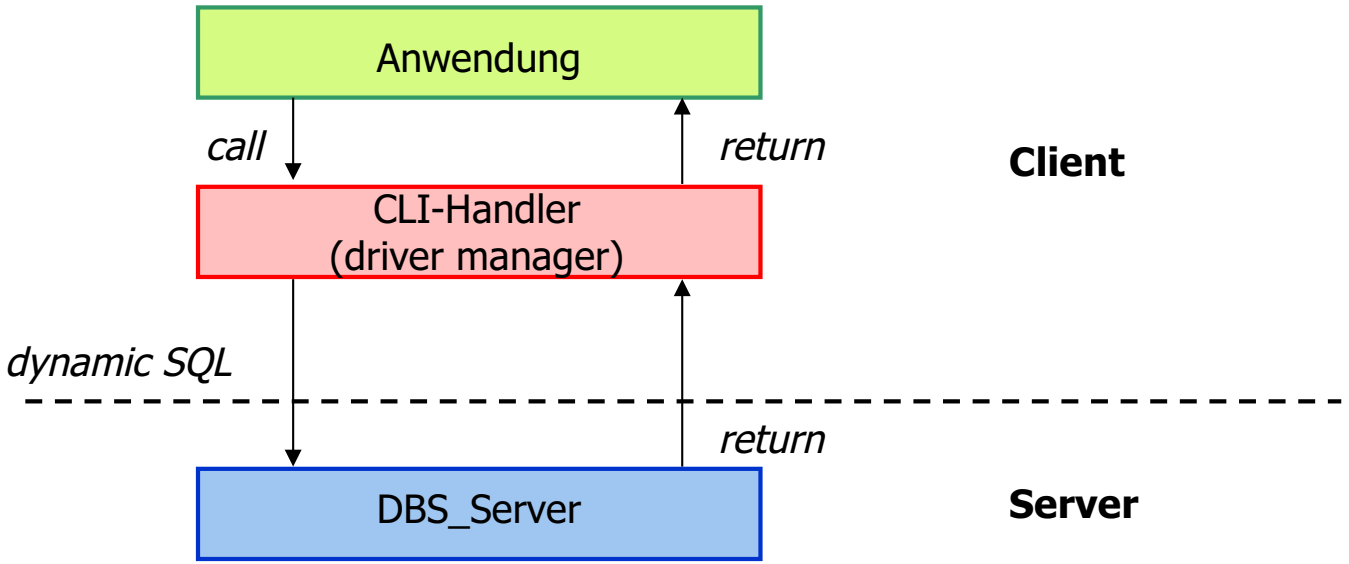
Wirtssprachen-Einbettung und Übersetzung

- Direkte Einbettung
  - keine syntaktische Unterscheidung zwischen Programm- und DB-Anweisungen
  - DB-Anweisung wird als Zeichenkette A ins AP integriert, z. B.
    exec sql open C1
  - verlangt Maßnahmen bei der AP-Übersetzung, typischerweise Einsatz eines Vorübersetzers PC (*Precompiler*)
- Aufruftechnik
  DB-Anweisung wird durch expliziten Funktionsaufruf an das Laufzeitsystem des DBMS übergeben, z. B.
  CALL DBS ('SELECT SUM(Gehalt) into :Gsum … ')
  - Es sind prinzipiell keine DBMS-spezifischen Vorkehrungen bei der AP-Übersetzung erforderlich!
  - Verschiedene Formen der Standardisierung:
    Call-Level-Interface (CLI), JDBC

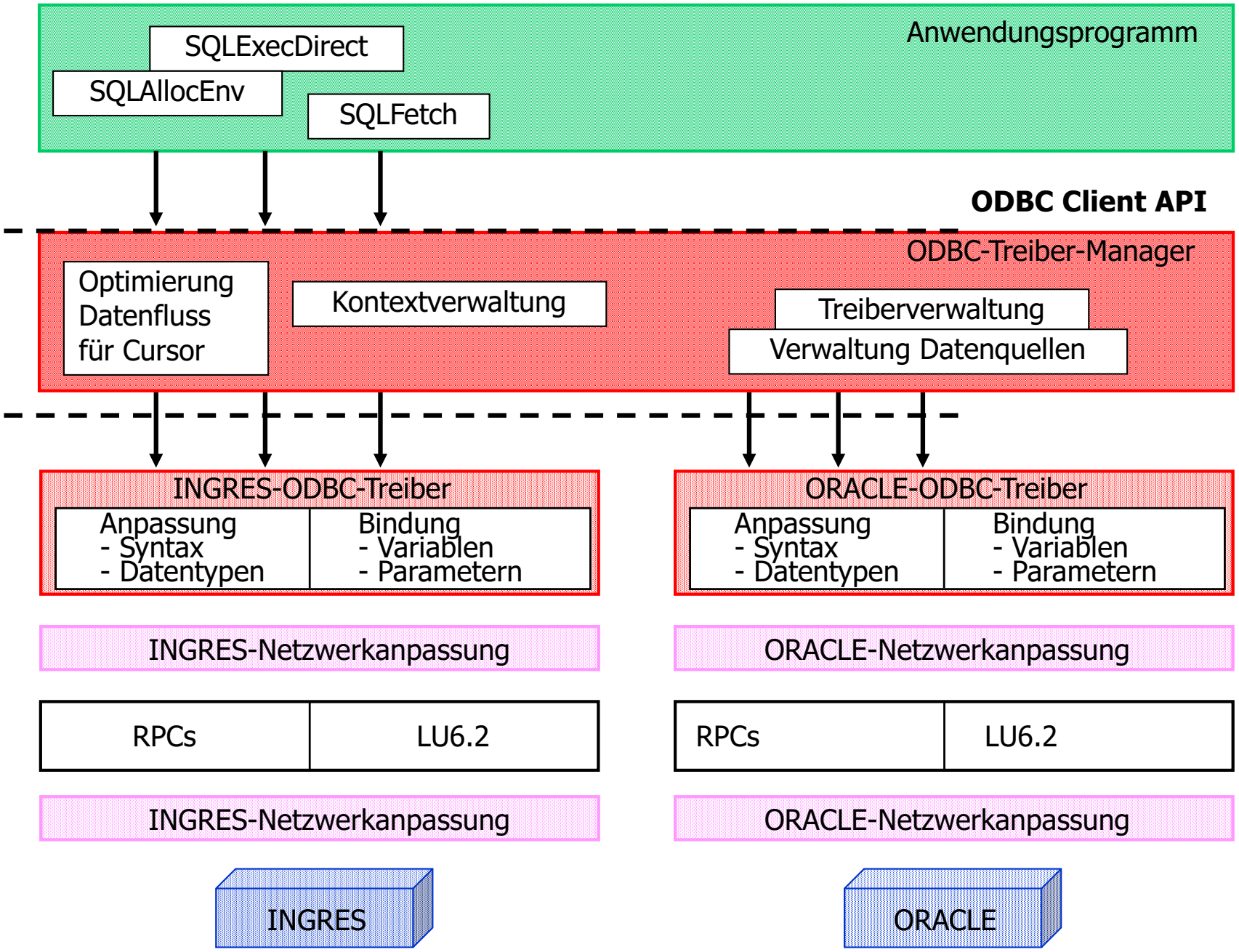
Call-Level-Interface

- Schnittstelle ist Sammlung von Prozeduren/Funktionen
  - Direkte Aufrufe der Routinen einer standardisierten Bibliothek
  - Keine Vorübersetzung (Behandlung der DB-Anweisungen)
    - Vorbereitung der DB-Anweisung geschieht erst beim Aufruf zur LZ
    - Anwendungen brauchen nicht im Quell-Code bereitgestellt werden
    - Wichtig zur Realisierung von kommerzieller AW-Software bzw. Tools

    ➔ Schnittstelle wird sehr häufig in der Praxis eingesetzt
- Einsatz typischerweise in Client/Server-Umgebung

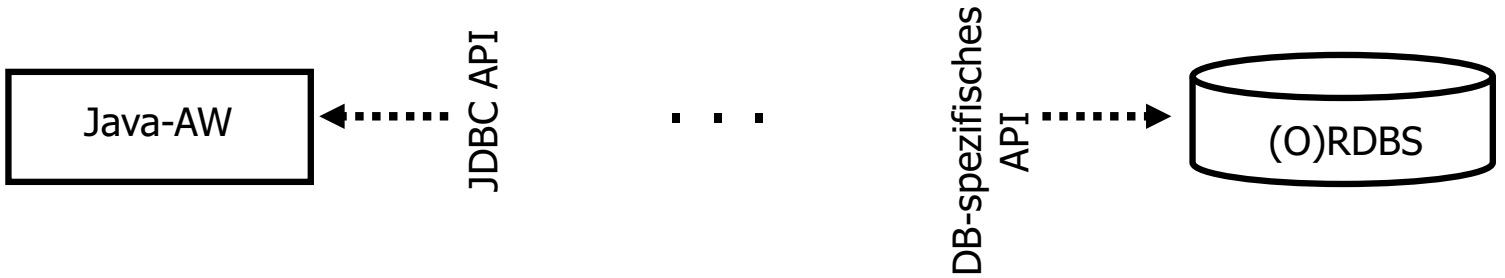


Beispiel Microsoft – ODBC-Architektur



DB-Zugriff via JDBC

- Java Database Connectivity Data Access API (JDBC)
  - unabhängiges, standardisiertes CLI, basierend auf SQL:1999
  - bietet Schnittstelle für den Zugriff auf (objekt-) relationale DBMS aus Java-Anwendungen
- Allgemeines Problem
  Verschiedene DB-bezogene APIs sind aufeinander abzubilden



- Überbrückung/Anpassung durch Treiber-Konzept
  - Treiber setzt JDBC-Aufrufe in die DBMS-spezifischen Aufrufe um
  - Treiber werden z.B. vom DBMS-Hersteller zur Verfügung gestellt

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 executeQuery koennen nun darauf Anfragen ausgefuehrt werden
12  // Ergebnisse in Form eines ResultSet Objekts
13  ResultSet rset = stmt.executeQuery(
          "SELECT p.persnr FROM professoren p") ;
```

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
25  // iteriere nun ueber Ergebnisse
26  while (rset.next( )) {
27        System.out.println(rset.getString(1));
28  }
```

JDBC Treiber für Postgresql

http://jdbc.postgresql.org/

Siehe insbesondere Dokumentation dazu (mit Beispielen), sowie ganz allgemein die Dokumentation zum Paket java.sql:

https://docs.oracle.com/javase/8/docs/api/index.html?java/sql/package-summary.html

JDBC – wichtige Funktionalität (1)

- Laden des Treiber
  - kann auf verschiedene Weise erfolgen, z.B. durch explizites Laden mit dem Klassenlader:

    ```
    Class.forName (DriverClassName)
    ```
- Aufbau und Schließen einer Verbindung
  - Connection-Objekt repräsentiert die Verbindung zum DB-Server
  - Beim Aufbau werden URL der DB, Benutzername und Paßwort als Strings übergeben:

    ```
    Connection con = DriverManager.getConnection (url, login, pwd);
    con.close();
    ```

JDBC – wichtige Funktionalität (2)

- Anweisungen
  - Mit dem Connection-Objekt können u.a. Metadaten der DB erfragt und Statement-Objekte zum Absetzen von SQL-Anweisungen erzeugt werden
  - Statement-Objekt erlaubt das Erzeugen einer SQL-Anweisung zur direkten (einmaligen) Ausführung

    ```
    Statement stmt = con.createStatement();
    ```

  - PreparedStatement-Objekt erlaubt das Erzeugen und Vorbereiten von (parametrisierten) SQL-Anweisungen zur wiederholten Ausführung

    ```
    PreparedStatement pstmt = con.prepareStatement (
            "select * from personal where gehalt >= ?");
    ```

  - Ausführung einer Anfrageanweisung speichert ihr Ergebnis in ein spezifiziertes ResultSet-Objekt

    ```
    ResultSet res = stmt.executeQuery ("select name from personal");
    ```

- Schließen von Statements

  ```
  Stmt.close();
  ```

JDBC - Anweisungen

- Anweisungen (Statements)
  - Sie werden in einem Schritt vorbereitet und ausgeführt
  - Sie entsprechen dem Typ EXECUTE IMMEDIATE im dynamischen SQL
  - JDBC-Methode erzeugt jedoch ein Objekt zur Rückgabe von Daten
- executeUpdate-Methode wird zur direkten Ausführung von UPDATE-, INSERT-, DELETE- und DDL-Anweisungen benutzt

```
Statement stat = con.createStatement ();
int n = stat.executeUpdate ("update personal
                             set gehalt = gehalt * 1.1
                             where gehalt < 5000.00");
// n enthält die Anzahl der aktualisierten Zeilen
```

- executeQuery-Methode führt Anfragen aus und liefert Ergebnismenge zurück

```
Statement stat1 = con.createStatement ();
ResultSet res1 = stat1.executeQuery (
    "select pnr, name, gehalt from personal where gehalt >=" + gehalt);
// Cursor-Zugriff und Konvertierung der DBMS-Datentypen in
// passende Java-Datentypen erforderlich (siehe Cursor-Behandlung)
```

JDBC – Prepared-Anweisungen

- PreparedStatement-Objekt

```
PreparedStatement pstmt;
double gehalt = 5000.00;
pstmt = con.prepareStatement (“select * from personal where gehalt >= ?”);
```

  - Vor der Ausführung sind dann die aktuellen Parameter einzusetzen mit Methoden wie setDouble, setInt, setString usw. und Indexangabe

```
pstmt.setDouble (1, gehalt);
```

  - Neben setXXX () gibt es Methoden getXXX () und updateXXX () für alle Basistypen von Java
- Ausführen einer Prepared-Anweisung als Anfrage

```
ResultSet res1 = pstmt.executeQuery ();
```

- Vorbereiten und Ausführung einer Prepared-Anweisung zur DB-Aktualisierung

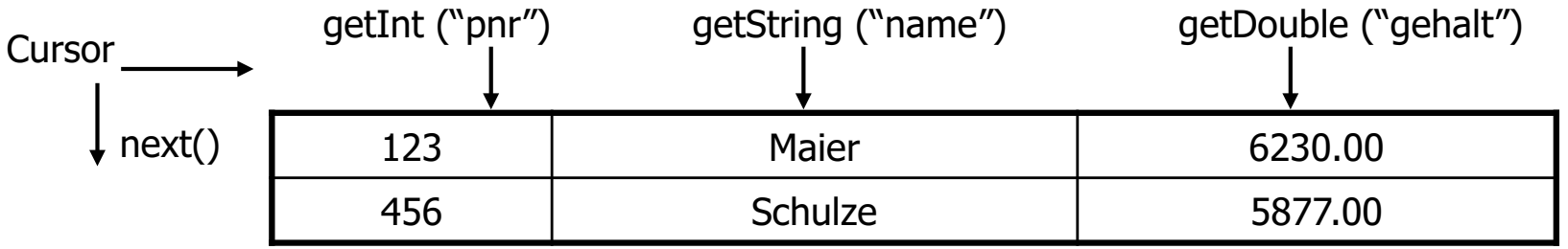
```
pstmt = con.prepareStatement (“delete from personal where name = ?”);
// set XXX-Methode erlaubt die Zuweisung von aktuellen Werten
pstmt.setString (1, “Maier”)
int n = pstmt.executeUpdate ();
// Methoden für Prepared-Anweisungen haben keine Argumente
```

JDBC – Ergebnismengen und Cursor

- Select-Anfragen und Ergebnisübergabe
  - Jede JDBC-Methode, mit der man Anfragen an das DBMS stellen kann, liefert ResultSet-Objekte als Rückgabewert

```
ResultSet res = stmt.executeQuery (
   "select pnr, name, gehalt from personal where gehalt ≥" +gehalt);
```

  - Cursor-Zugriff und Konvertierung der DBMS-Datentypen in passende Java-Datentypen erforderlich
  - JDBC-Cursor ist durch die Methode next() der Klasse ResultSet implementiert



  - Zugriff aus Java-Programm

```
while (res.next() ) {
      System.out.print (res.getInt ("pnr") + "\t");
      System.out.print (res.getString ("name") + "\t");
      System.out.println (res.getDouble ("gehalt") );
}
```

JDBC – Ergebnismengen und Cursor (2)

- JDBC definiert drei Typen von ResultSets

  ➔ siehe Cursor-Eigenschaften
- ResultSet: forward-only
  - Default-Cursor vom Typ INSENSITIVE: nur next()
- ResultSet: scroll-insensitive
  - Scroll-Operationen sind möglich, aber DB-Aktualisierungen verändern ResultSet nach seiner Erstellung nicht
- ResultSet: scroll-sensitve
  - Scroll-Operationen sind möglich, wobei ein nicht-INSENSITIVE Cursor benutzt wird
  - Semantik der Operation, im Standard nicht festgelegt, wird vom DBMS-Hersteller definiert!
  - Oft wird ODBCs sog. KEYSET_DRIVEN-Semantik implementiert.
    - Änderungen und Löschungen nachträglich sichtbar, Einfügungen nicht!

JDBC – Ergebnismengen und Cursor (3)

- Aktualisierbare ResultSets

```
Statement s1 = con1.createStatement (
      ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet res= s1.executeQuery (. . .);    . . .
res.updateString (“name”, “Müller”);    . . .
res.updateRow ();
```

  - Zeilen können in aktualisierbaren ResultSets geändert und gelöscht werden.
  - Mit res.insertRow () wird eine Zeile in res und gleichzeitig auch in die DB eingefügt.

JDBC – Zugriff auf Metadaten

- Allgemeine Metadaten
  - Welche Information benötigt ein Browser, um seine Arbeit beginnen zu können?
  - JDBC besitzt eine Klasse DatabaseMetaData, die zum Abfragen von Schema- und anderer DB-Information herangezogen wird
- Informationen über ResultSets
  - JDBC bietet die Klasse ResultSetMetaData

    ```
    ResultSet rs1 = stmt1.executeQuery (“select * from personal”);
    ResultSetMetaData rsm1 = rs1.getMetaData ();
    ```
  - Es müssen die Spaltenanzahl sowie die einzelnen Spaltennamen und ihre Typen erfragt werden können (z. B. für die erste Spalte)

    ```
    int AnzahlSpalten = rsm1.getColumnCount ();
    String SpaltenName = rsm1.getColumnName (1);
    String TypName = rsm1.getColumnTypeName (1);
    ```
  - Ein Wertzugriff kann dann erfolgen durch

    `rs1.getInt (2)`, wenn

    `rsm1.getColumnTypeName (2)`

    den String “Integer” zurückliefert.

JDBC – Fehler

- Fehlerbehandlung
  - Spezifikation der Ausnahmen, die eine Methode werfen kann, bei ihrer Deklaration (throw exception)
  - Ausführung der Methode in einem try-Block, Ausnahmen werden im catch-Block abgefangen

```
try   {
. . . Programmcode, der Ausnahmen verursachen kann
}
catch (SQLException e) {
        System.out.println (“Es ist ein Fehler aufgetreten :\n”);
        System.out.println (“Msg: “ + e.getMessage () );
        System.out.println (“SQLState: “ + e.getSQLState () );
        System.out.println (“ErrorCode: “ + e.getErrorCode () );
};
```

JDBC – Transaktionen

- Transaktionen
  - Bei Erzeugen eines Connection-Objekts (z.B. con1) ist als Default der Modus autocommit eingestellt
  - Um Transaktionen als Folgen von Anweisungen abwickeln zu können, ist dieser Modus auszuschalten

    ```
    con1.setAutoCommit(false);
    ```
  - Für eine Transaktion können sogen. Konsistenzebenen (isolation levels) wie TRANSACTION_SERIALIZABLE, TRANSACTION_REPEATABLE_READ usw. eingestellt werden

    ```
    con1.setTransactionIsolation (Connection.TRANSACTION_SERIALIZABLE);
    ```
- Beendigung oder Zurücksetzen

  ```
  con1.commit();
  con1.rollback();
  ```
- Programm kann mit mehreren DBMS verbunden sein
  - selektives Beenden/Zurücksetzen von Transaktionen pro DBMS
  - kein globales atomares Commit möglich

DB-Zugriff via JDBC – Beispiel 1

```
import java.sql.*;
public class Select   {
     public static void main (String [ ] args) {
        Connection con = null;
        PreparedStatement pstmt;
        ResultSet res;
        double gehalt = 5000.00;
        try   {
           Class.forName (“sun.jdbc.odbc.JdbcOdbcDriver”);
           con = java.sql.DriverManager.getConnection (“jdbc:odbc:personal”, “user”, “passwd”);
           pstmt = con.prepareStatement (“select pnr, name, gehalt from personal where gehalt >= ?”);
           pstmt.setDouble (1, gehalt);
           . . .
           res = pstmt.executeQuery ();
           while (res.next () )   {
                    System.out.print (res.getInt (“pnr”) + “\t”);
                    System.out.print (res.getString (“name”) + “\t”);
                    System.out.printIn (res.getDouble (“gehalt”) );
           }
           res.close ();
           pstmt.close ();
        } / / try
           catch (SQLException e)     {
                    System.out.printIn (e);
                    System.out.printIn (e.getSQLState () );
                    System.out.printIn (e.getErrorCode () );
           }
           catch (ClassNotFoundException e)     {
                    System.out.printIn (e);
           }
        } / / main
     } / / class Select
```

DB-Zugriff via JDBC – Beispiel 2

```
import java.sql.*;
public class Insert   {
     public static void main (String [ ] args)    {
        Connection con = null;
        PreparedStatement pstmt;
        try {
           Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
           con = java.sql.DriverManager.getConnection ("jdbc.odbc.personal", " ", " ");
           pstmt = con.prepareStatement ("insert into personal values (?, ?, ?)");
           pstmt.setInt (1, 222);
           pstmt.setString (2, "Schmitt");
           pstmt.setDouble (3, 6000.00);
           pstmt.executeUpdate ();
           pstmt.close ();
           con.close ();
        } / / try
           catch (SQLException e)    {
                    System.out.printIn (e);
                    System.out.printIn (e.getSQLState () );
                    System.out.printIn (e.getErrorCode () );
           }
           catch (ClassNotFoundException e) {System.out.printIn (e);
           }
         }
      }
. . .
           pstmt = con.prepareStatement ("update personal set gehalt = gehalt * 1.1 where gehalt < ?");
           pstmt.setDouble (1, 10000.00);
           pstmt.executeUpdate ();
           pstmt.close ();
. . .
           pstmt = con.prepareStatement ("delete from personal where pnr = ?");
           pstmt.setInt (1,222);
           pstmt.executeUpdate ();
           pstmt.close ();
```

Anmerkung: SQL Injections

- Beispiel
  - SQL Anfrage wird in Anwendung erstellt, wobei id eine Benutzereingabe ist

    …. "SELECT author, subject, text" +
    "FROM artikel WHERE ID=' " + id + " ' "
  - Aufruf z.B. durch Webserver http://webserver/cgi-bin/find.cgi?ID=42
- SQL Injection zum Ausspähen von Daten

  http://webserver/cgi-bin/find.cgi?ID=42'+UNION+SELECT+
  login,+password,+'x'+FROM+user+WHERE+'x'='x
  - Führt zur SQL Anweisung:

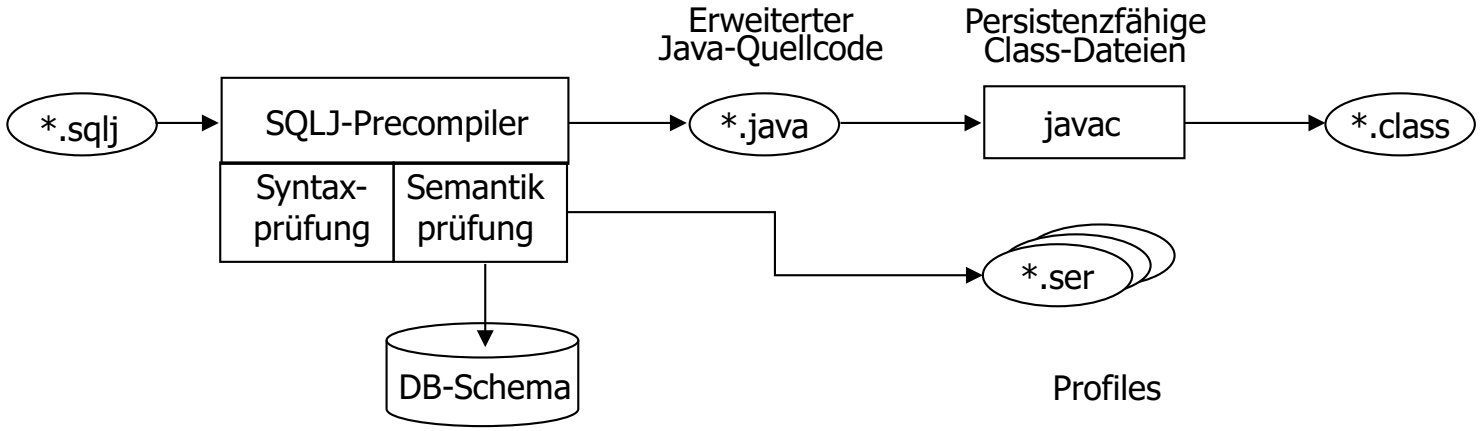
    select author, subject, text from artikel
    where ID='42' union select login, password, 'x' from user where 'x'='x';
  - Und andere Fälle bis hin zum Einschleusen von beliebigem Code auf Rechner + Öffnen einer Shell (abh. von DBMS)
- Hilfe u.a. durch Benutzen von PreparedStatements mit Parametern!
  - Übersicht unter: http://de.wikipedia.org/wiki/SQL-Injection

SQLJ – Eingebettetes SQL in Java

- SQLJ (offiziell: SQL/OLB – *Object Language Bindings*)
  - beschreibt die Einbettung von SQL in Java-Anwendungen
  - besitzt bessere Lesbarkeit, Verständlichkeit und Wartbarkeit durch kurze und prägnante Schreibweise
  - zielt auf die Laufzeiteffizienz von eingebettetem SQL ab, ohne die Vorteile des DB-Zugriffs via JDBC aufzugeben
- SQLJ und JDBC
  - Ebenso wie statische und dynamische SQL-Anweisungen in einem Programm benutzt werden können, können SQLJ-Anweisungen und JDBC-Aufrufe im selben Java-Programm auftreten.
- Im Folgenden werden nur einige Unterschiede zu eingebettetem SQL und JDBC aufgezeigt

SQLJ - Abbildung auf JDBC durch Precompiler

- Abbildung auf JDBC durch Precompiler
  - Überprüfung der Syntax sowie (gewisser Aspekte) der Semantik von SQL-Anweisungen (Anzahl und Typen von Argumenten usw.) zur Übersetzungszeit, was Kommunikation mit dem DBMS zur Laufzeit erspart
  - Ersetzung der SQLJ-Anweisungen durch Aufrufe an das SQLJ-Laufzeitmodul (Package sqlj.runtime.*)
  - Erzeugung sog. Profiles, serialisierbare Java-Klassen, welche die eigentlichen JDBC-Anweisungen enthalten
  - Abwicklung von DB-Anweisungen vom SQLJ-Laufzeitmodul dynamisch über die Profiles, die wiederum über einen JDBC-Treiber auf die DB zugreifen
  - Anpassung an ein anderes DBMS geschieht durch Austausch der Profiles (sog. Customizing)



SQLJ ConnectionContext

- Verbindung zum DBMS
  - erfolgt über sog. Verbindungskontexte (ConnectionContext)
  - Sie basieren auf JDBC-Verbindungen und werden auch so genutzt (URL, Nutzername, Paßwort)
  - SQLJ-Programm kann mehrere Verbindungskontexte über verschiedene JDBC-Treiber aufbauen; sie erlauben den parallelen Zugriff auf mehrere DBMS oder aus mehreren Threads/Prozessen auf das gleiche DBMS

SQLJ – Anweisungen und Iteratoren

- SQL-Anweisungen sind im Java-Programm Teil einer SQLJ-Klausel

```
#SQL { select p.onr into :persnr
       from personal p
       where p.beruf = :beruf
       and p.gehalt > :gehalt};
```

  - Austausch von Daten zwischen SQLJ und Java-Programm erfolgt über Wirtssprachenvariablen
  - Parameterübergabe kann vorbereitet werden
  - ist viel effizienter als bei JDBC (mit ?-Platzhaltern)
- Iteratoren
  - analog zu JDBC-ResultSets
  - Definition von Iteratoren (Cursor), aus denen entsprechende Java-Klassen generiert werden, über die auf die Ergebnismenge zugegriffen wird

SQLJ NamedIterators

- Nutzung eines Iterators in SQLJ

```
import java.sql.*
. . .
#SQL iterator GetPersIter (int personalnr, String nachname);
Get PersIter iter1;
#SQL iter1 = {select  p.pnr as "personalnr", p.name as "nachname"
                    from personal p
                    where      p.beruf = :Beruf
                    and        p.gehalt = :Gehalt};
int Id;
String Name;
while      (iter1.next ())  {
           Id = iter1.personalnr ();
           Name = iter1.nachname ();
           … Verarbeitung …
}
iter1.close ();
```

- SQLJ liefert für eine Anfrage ein SQLJ-Iterator-Objekt zurück
  - SQLJ-Precompiler generiert Java-Anweisungen, die eine Klasse GetPersIter definieren
  - Deklaration gibt den Spalten Java-Namen (personalnr u. nachname) und definiert implizit Zugriffsmethoden personalnr() u. nachname(), die zum Iterator-Zugriff benutzt werden

Zusammenfassung - DB-Zugriff in Programmen

- Cursor-Konzept zur satzweisen Verarbeitung von Datenmengen
  - Anpassung von mengenorientierter Bereitstellung und satzweiser Verarbeitung von DBMS-Ergebnissen
  - Operationen: DECLARE CURSOR, OPEN, FETCH, CLOSE
- Statisches (eingebettetes) SQL (ESQL)
  - hohe Effizienz, gesamte Typprüfung und Konvertierung erfolgen durch Precompiler
  - relativ einfache Programmierung
  - Aufbau aller SQL-Befehle muss zur Übersetzungszeit festliegen
  - es können zur Laufzeit nicht verschiedene Datenbanken dynamisch angesprochen werden
- Call-Level Interface
  - beliebige Statements können dynamisch erzeugt werden
  - kein Precompiler, dadurch erhöhte Portabilität
- DB-Zugriff in Java-Anwendungen
  - CLI: JDBC
  - ESQL: SQLJ – aber trotzdem Portabilität!

Benutzerdefinierte Routinen in SQL

- 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 aus dem DBMS zur Anwendung gebracht werden (und umgekehrt)
  - Höhere Performance
  - Code kann wiederverwendet werden (zwischen Anwendungen)
- Nachteile
  - Etwas aufwendiger zu erstellen.
  - Debugging schwieriger.

Benutzerdefinierte Routinen in SQL (2)

- Zwei Arten von Routinen
  - Prozedur (*stored procedure*)
    - wird mit CREATE PROCEDURE … erzeugt
    - kann Ergebnisse mit Hilfe von sog. OUT/INOUT-Parametern zurückgeben
    - kann auch Resultatsmengen (result sets) zurückgeben
    - wird mit dem CALL-Befehl in SQL aufgerufen
  - Funktion (*user-defined function, UDF*)
    - wird mit CREATE FUNCTION … erzeugt
    - liefert immer einen Ergebniswert zurück
    - wird über Funktionsaufruf in anderen SQL-Befehlen (z.B. SELECT, UPDATE, ...) ausgeführt
- Implementierungsalternativen
  - in SQL, mit prozeduralen Erweiterungen (sog. PSM)
  - in einer (externen) Programmiersprache (z.B. in C oder Java)
    - Programmcode kann wieder SQL-Befehle enthalten (z.B. JDBC, SQLJ)

SQL-PSM: Prozedurale Spracherweiterungen

▪ Compound statement	BEGIN ... END;
▪ SQL variable declaration	DECLARE var CHAR (6);
▪ If statement	IF var <> 'urgent' THEN ... ELSEIF … ELSE ...;
▪ Case statement	CASE var WHEN 'SQL' THEN < SQL statement list> WHEN ...;
▪ Loop statement	LOOP < SQL statement list> END LOOP;
▪ While statement	WHILE i<100 DO .... END WHILE;
▪ Repeat statement	REPEAT ... UNTIL i>=100 END REPEAT;
▪ For statement	FOR result AS ... DO ... END FOR;
▪ Leave statement	LEAVE ...;
▪ Return statement	RETURN 'urgent';
▪ Call statement	CALL procedure_x (1,3,5);
▪ Assignment statement	SET x = 'abc';
▪ Signal/resignal statement	SIGNAL division_by_zero

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.

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.

Beispiel (in PL/pgSQL)

```
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

Routinen in Postgresql

- Postgresql
  - In Postgres gibt es keinen Unterschied zwischen stored procedures und UDFs
  - EXEC CALL gibt es in Postgres nicht.
  - UDF wird aufgerufen in SELECT statements, z.b. **SELECT FROM** myFunction(44234234);
- 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/)

https://www.postgresql.org/docs/current/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 im Beispiel 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)

- Die Funktion im Beispiel unten 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)

- Rückgabewerte: 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 ] ;
```

```
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 zurückliefern, 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:

`https://www.postgresql.org/docs/current/static/plpgsql-statements.html`

PL/pgSQL: Kontrollstrukturen

- PL/pgSQL bietet die übliche Auswahl an 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;
```

Funktionen als Triggeraktionen in Postgresql

- Triggeraktionen immer als Prozedur/Funktion implementiert.

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

```
1   CREATE OR REPLACE
2       FUNCTION pflichtvorlesungLogikfunction( )
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: Parameterübergabe an Funktion

- Es macht natürlich wenig Sinn für jede Pflichtvorlesung eine eigene Funktion zu schreiben.
- Besser ist es, 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;
```

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