Kapitel 8: SQL-Einbettung in Programmiersprachen

Kopplungsvarianten zwischen Programmiersprachen und Datenbanksprachen:

1) Erweiterung der Programmiersprache um Datenbankkonstrukte
   → Persistente Programmiersprachen, Datenbankprogrammiersprachen
   (z.B. Pascal/R, DBPL, Persistentes C++)
2) Erweiterung der Datenbanksprache um Programmierkonstrukte
   → Skriptsprachen für Stored Procedures, "4th Generation Languages" (4GLs)
3) Einbettung der Datenbanksprache in die Programmiersprache (oder Web-Skriptsprache)
   → "Embedded SQL" (ESQL)

Beispiel für eine Datenbankprogrammiersprache (à la DBPL im Pascal- bzw. Modula-Stil):

```
TYPE    Produktrecordtyp        = RECORD
                                    PNR: INTEGER;
                                    ...
                                  END;
        Produkterelationentyp   = RELATION OF Produktrecordtyp;

VAR     Produkte: PERSISTENT Produkterelationentyp;
        Ergebnis: Produkterelationentyp;
        x: Produktrecordtyp;

Ergebnis :=  SELECT *
             FROM Produkte
             WHERE ...;

FOR EACH x IN Ergebnis
DO
   ...
END;
```

8.1 Eingebettetes SQL (Embedded SQL)

Kernproblem bei der SQL-Einbettung in konventionelle Programmiersprachen:

Abbildung von Tupelmengen auf die Datentypen der Wirtsprogrammiersprache

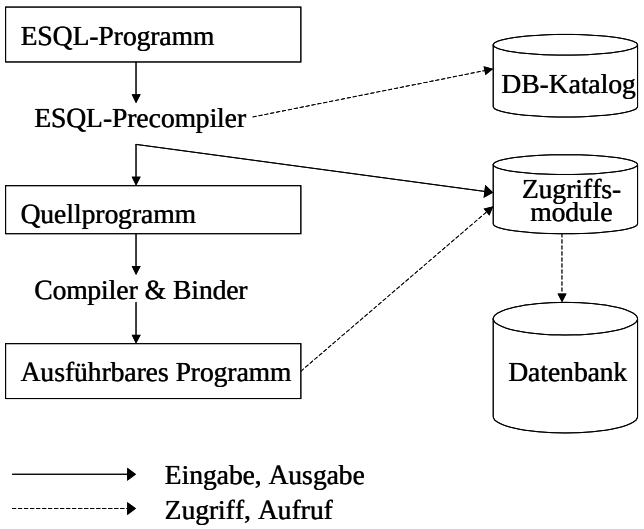
Realisierte Lösung:

Abbildung von Tupeln bzw. Attributen auf die Datentypen der Programmiersprache
→ Wirtsprogrammvariable (engl.: Host Variables)

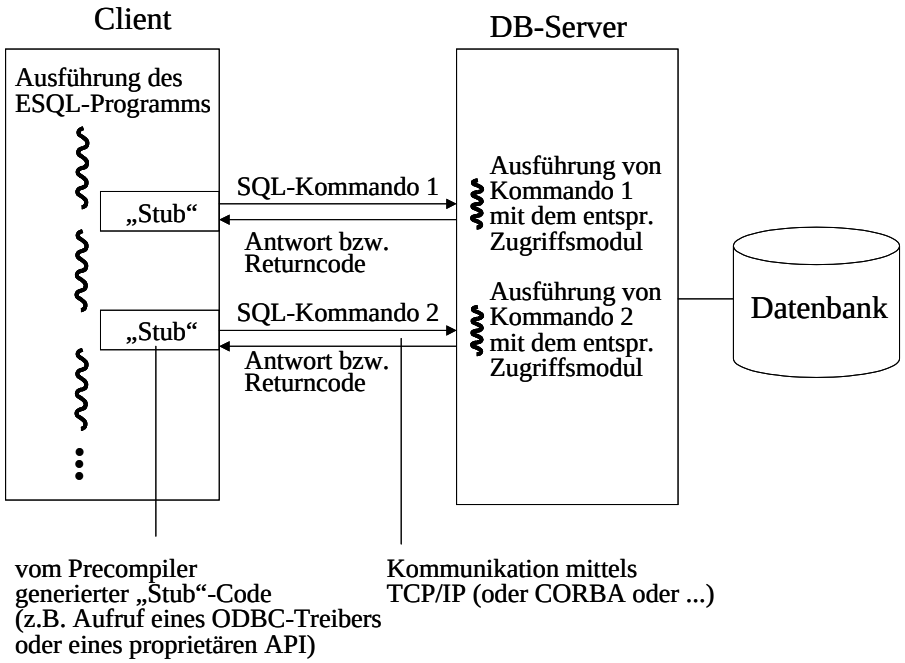
+

Iteratoren (Schleifen) zur Verarbeitung von Tupelmengen
→ Cursor-Konzept

Standardarchitektur:



Laufzeitarchitektur für ESQL-Programme in einem Client-Server-System:



Wirtsprogrammvariable (Host Variables)

Die Attribute eines Resultattupels einer SQL-Anfrage werden an speziell deklarierte Variable des Wirtsprogramms zugewiesen (INTO-Klausel).
Beispiel:

```
EXEC SQL    SELECT PNr, Menge, Status INTO :pnr, :menge, :status
            FROM Bestellungen WHERE BestNr = 555;
```

Analog können SQL-Anweisungen mittels solcher Variable mit Eingabeparametern versorgt werden.
Beispiele:

```
1) EXEC SQL    SELECT PNr, Menge, Status INTO :pnr, :menge, :status
               FROM Bestellungen WHERE BestNr = :bestnr;
2) EXEC SQL    INSERT INTO Bestellungen (BestNr, Monat, Tag, KNr, PNr, Menge)
               VALUES (:bestnr, :monat, :tag, :knr, :pnr, :menge);
```

Wirtsprogrammvariable dienen als "Übergabepuffer" zwischen DBS und Programm. Generell werden dabei die SQL-Datentypen auf die Datentypen der jeweiligen Wirtsprogrammiersprache abgebildet.

Indikatorvariable zum Erkennen von Nullwerten

Beispiel:

```
EXEC SQL BEGIN DECLARE SECTION;
   int      pnr;
   int      vorrat;
   short    vorrat_ind;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL    SELECT Vorrat INTO :vorrat:vorrat_ind
            FROM Produkte WHERE PNr = :pnr;
if (vorrat_ind = = 0)
   { /* kein Nullwert */ ... }
else
   { /* Nullwert */ ... };
```

Allgemein können Indikatorvariable folgende Werte haben:

= 0	die entsprechende Wirtsprogrammvariable hat einen regulären Wert
= -1	die entsprechende Wirtsprogrammvariable hat einen Nullwert
> 0	die entsprechende Wirtsprogrammvariable enthält eine abgeschnittene Zeichenkette

Beispiel eines ESQL-Programms (lieferung.pc)

```
/*** Programm zur Erfassung von Lieferungen ***/

#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA; /* Importieren der SQL Communication Area */
/*  Innerhalb der SQL-Anweisungen ist Gross-/Kleinschreibung irrelevant.
    Sie dient hier nur zur Hervorhebung der SQL-Anweisungen          */

/* Der Precompiler erzeugt an dieser Stelle die folgende Datenstruktur:
    struct sqlca {
        char    sqlcaid[8];
        long    sqlabc;
        long    sqlcode;
        struct  {
                unsigned short  sqlerrml;
                char            sqlerrmc[70];
                } sqlerrm;
        char    sqlerrp[8];
        long    sqlerrd[6];
        char    sqlwarn[8];
        char    sqlext[8];
        };
    struct sqlca sqlca;
*/

main ()
{
/* Deklaration der Wirtsprogrammvariablen */
EXEC SQL BEGIN DECLARE SECTION;
    /*  Die hier deklarierten Wirtsprogrammvariablen unterliegen bei ihrer
        Verwendung in SQL-Ausdrücken einer Typprüfung durch den Precompiler. */
    int bestnr;
    int pnr;
    int menge;
    VARCHAR status[10];

    /* Der Precompiler erzeugt aus einer solchen Deklaration die folgende Struktur:
        struct {
                unsigned short len;
                unsigned char arr[10];
                } status;
    */

    VARCHAR user[20];
    VARCHAR passwd[10];
EXEC SQL END DECLARE SECTION;

/* Globale Fehlerbehandlung */
EXEC SQL WHENEVER SQLERROR STOP;
```

```
/* Verbindungsaufbau mit dem DBS */
printf ("Benutzername?"); scanf ("%s", &(user.arr));
user.len = strlen(user.arr); /* Konvertierung von C-String in VARCHAR */
printf ("Kennwort?"); scanf ("%s", &(passwd.arr));
passwd.len = strlen(passwd.arr); /* Konvertierung von C-String in VARCHAR */
EXEC SQL CONNECT :user IDENTIFIED BY :passwd;
/* gleichzeitig Beginn einer Transaktion */

/* Dialogschleife */
printf ("Bitte geben Sie eine Bestellnummer ein. (0 = Programmende)\n");
scanf ("%d", &bestnr);

while (bestnr != 0)
   {
   EXEC SQL    SELECT PNr, Menge, Status
               INTO :pnr, :menge, :status
               FROM Bestellungen
               WHERE BestNr = :bestnr;
   if (sqlca.sqlcode = = 0) /* Testen des SQL-Returncodes */
      {
      status.arr[status.len] = '\0'; /* Konvertierung von VARCHAR in C-String */
      if (strcmp (status.arr, "neu") = = 0)
         {
         EXEC SQL    UPDATE Produkte
                     SET Vorrat = Vorrat - :menge
                     WHERE PNr = :pnr AND Vorrat >= :menge;
         if (sqlca.sqlcode = = 0) /* Testen des SQL-Returncodes */
            {
            strcpy (status.arr, "geliefert");
            status.len = strlen (status.arr);
            EXEC SQL    UPDATE Bestellungen
                        SET Status = :status
                        WHERE BestNr = :bestnr;
            }
         else
            printf ("\n *** Ungenuegender Lagervorrat ***\n");
         }
      else
         printf ("\n *** Lieferung bereits erfolgt ***\n");
      }
   else
      printf ("\n *** Bestellung nicht gefunden ***\n");
   EXEC SQL COMMIT WORK;  /* Ende Transaktion und Beginn neue Transaktion */
   printf ("Bitte geben Sie eine Bestellnummer ein. (0 = Programmende)\n");
   scanf ("%d", &bestnr);
   }; /*while*/

EXEC SQL DISCONNECT;    /* Verbindung mit dem DBS abbauen */
exit (0);
} /*main*/
```

Fehlerbehandlung in ESQL

1) Explizites Testen von sqlca.sqlcode im Wirtsprogramm nach einer SQL-Anweisung:

= 0 → Anweisung korrekt ausgeführt, keine besonderen Vorkommnisse
< 0 → Fehler bei der Ausführung (siehe Fehlercodes im Manual)
> 0 → Anweisung ausgeführt, Auftreten eines Sonderfalls,
z.B. signalisiert der Wert 1403, daß keine (weiteren) Treffertupel existieren

Zusatzinformation in den restlichen Komponenten von sqlca, z.B.:

sqlca.sqlerrd[2]	Anzahl der Tupel, die von einer Insert-, Update- oder Delete-Anweisung betroffen waren
sqlca.sqlerrm.sqlerrmc	Fehlermeldung als Ascii-Text (max. 70 Zeichen)
sqlca.sqlerrm.sqlerrml	Länge der Fehlermeldung

2) Deklaration von "Exception-Handling"-Strategien:

```
EXEC SQL WHENEVER    ( SQLERROR | NOT FOUND | SQLWARNING )
                        ( STOP | GOTO label | CONTINUE )
```

wobei
SQLERROR einem SQLCODE < 0 entspricht,
NOT FOUND dem SQLCODE 1403 und
SQLWARNING einem SQLCODE > 0 (aber ungleich 1403).

Der ESQL-Precompiler erzeugt automatisch nach jeder SQL-Anweisung einen entsprechenden Vergleich mit sqlca.sqlcode, und zwar jeweils aufgrund der *textuell* letzten WHENEVER-Anweisung vor der SQL-Anweisung. Extrem schwerwiegende Fehler führen immer zum sofortigen Abbruch.

Achtung: Die WHENEVER-Klausel hat potentielle Fallen.

1) Im folgenden Programm ist in f die WHENEVER-Spezifikation von g nicht wirksam.

```
void f (...) ...
   { ...  EXEC SQL UPDATE ...; ... };
void g (..) ...
   { ...  EXEC SQL WHENEVER SQLERROR GOTO handle_error; ...
          f (...); ...
   };
```

2) Das folgende Programm führt u.U. zu einer Endlosschleife.

```
...
EXEC SQL WHENEVER SQLERROR GOTO handle_error;
EXEC SQL CREATE TABLE Mytable ( ... );
...
handle_error:
   EXEC SQL DROP TABLE Mytable;
   EXEC SQL DISCONNECT;
   exit (1);
```

Korrektur:
Im Teil hinter der Marke "handle_error" als erstes spezifizieren:
EXEC SQL WHENEVER SQLERROR CONTINUE;

Verarbeitung von Tupelmengen mittels Cursor-Konzept

Zweck:
Verarbeitung von Tupelmengen in einer nichtmengenorientierten Wirtsprogrammiersprache.
Notwendig für alle Anfragen mit mehr als einem Resultattupel.

Beispiel: Ausgabe aller Bestellungen eines Kunden

```
#define TRUE 1
#define FALSE 0
...
printf ("Bitte geben Sie eine Kundennummer ein. (0 = Programmende)\n");
scanf ("%d", &knr);
EXEC SQL DECLARE Kundeniterator CURSOR FOR
    SELECT Monat, Tag, Bez, Menge
    FROM Bestellungen WHERE KNr = :knr
    ORDER BY Monat DESC, Tag DESC;
/* Cursor-Deklaration immer ohne INTO-Klausel */
...
EXEC SQL OPEN Kundeniterator;
/*  An dieser Stelle werden die Eingabeparameter der SQL-Anweisung (:knr) ausgewertet,
    und gedanklich wird hier die Resultattupelmenge ermittelt.                    */
found = TRUE;
while (found) /* solange es noch Resultattupel gibt */
    {
    EXEC SQL FETCH Kundeniterator INTO :monat, :tag, :bez, :menge;
    /* Hier werden die Wirtsprogrammvariable für die Resultattupel festgelegt. */
    bez.arr[bez.len] = '\0';
    if ((sqlca.sqlcode >= 0) && (sqlca.sqlcode != 1403))
        printf ("%d.%d.: %d %s", tag, monat, menge, bez);
    else
        found = FALSE;
    }; /*while*/
EXEC SQL CLOSE Kundeniterator;
/* Freigabe des Cursors und der damit verbundenen DBS-Ressourcen */
```

8.2 Dynamisches ESQL

Zweck:

Einbettung von SQL-Anweisungen, deren Struktur erst zur Laufzeit des Wirtsprogramms bekannt ist.

Bei Einbettung von SQL in die (auf Bytecodebene interpretierte) Programmiersprache Java mit JDBC als Datenbankzugriffsprotokoll (einer Erweiterung von ODBC) ist dynamisches SQL der Regelfall. Die Vorübersetzung von SQL-Anweisungen im Java-Programm ist im Standard SQLJ vorgesehen, bislang aber wenig verbreitet.

Einfacher Fall: Vorgehensweise bei statisch festgelegtem Resultatschema und statisch festgelegten Eingabeparametern:

Beispiel ("dynamischer Anteil" ist unterstrichen):

```
SELECT Monat, Tag, Bestellungen.PNr, Bez, Menge
FROM Bestellungen, Produkte WHERE Bestellungen.PNr = Produkte.PNR
AND ( Bez LIKE 'Druck%' OR Bez LIKE 'Papier%' OR Bez LIKE 'Platte%' )
ORDER BY Monat DESC, Tag DESC
```

Vorgehensweise:

1) Deklaration der Wirtsprogrammvariablen
2) Aufbau der SQL-Anweisung als Zeichenkette
3) Dynamische Precompilation: PREPARE DynQuery FROM :sqltext
4) DECLARE Iterator CURSOR FOR DynQuery
5) Auswertung der Eingabeparameter und Vorbereitung der "Cursor"-Schleife:
   OPEN Iterator
6) FETCH Iterator INTO Wirtsprogrammvariablen
7) Resultatsattribute in den Wirtsprogrammvariablen verarbeiten
8) Wiederholung der Schritte 6 und 7
   für jedes Resultatstupel
9) CLOSE Iterator

Sonderfall: Anfrageresult hat höchstens ein Tupel
(z.B. bei Anfragen über den Primärschlüssel oder bei Änderungsoperationen)
statt 3) bis 9) einfacher:
EXECUTE IMMEDIATE :sqltext

Komplexer Fall: Vorgehensweise bei dynamisch festgelegtem Resultatschema oder dynamisch festgelegten Eingabeparametern (mit dynamischer Allokation der Datenübergabebereiche und dynamischer Übersetzung für wiederholte Ausführung):

Abfrage der Typbeschreibung mit DESCRIBE queryname INTO ptr-to-sqldescriptorarea,
dynamische Allokation der Übergabebereiche im Programm und
Eintragen entsprechender Pointer in die sqldescriptorarea (SQLDA) vor der Queryausführung.
(siehe Handbücher; für Vorlesung nicht relevant)

Beispiel für den einfachen Fall:

```
/***  Programm zum Ausdrucken der Bestellungen fuer eine Liste von Produkten ***/
/***  Eingabe: Liste von Produktbezeichnungen                                ***/
/***  Ausgabe: Bestellungen dieser Produkte, absteigend sortiert nach Datum  ***/

#include <stdio.h>
#include <string.h>
#define TRUE 1
#define FALSE 0
#define MAXPRODUKTE 10
/* Die Bestellungen umfassen maximal 10 Produkte */
EXEC SQL INCLUDE SQLCA; /* Importieren der SQLCA */

main () {
EXEC SQL BEGIN DECLARE SECTION;
    int monat, tag, pnr, menge;
    VARCHAR bez[31];
    VARCHAR sqltext[512];
    VARCHAR user[20];
    VARCHAR passwd[10];
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR GOTO handle_error;

/* Typdeklarationen */
typedef char prod_bez_t[31];
typedef int bool;

/* Variablendeklarationen */
prod_bez_t     prod_bez[10];
int            i;
int            num_of_prod;
bool           found;

printf ("Benutzername?"); scanf ("%s", &(user.arr)); user.len = strlen(user.arr);
printf ("Kennwort?"); scanf ("%s", &(passwd.arr)); passwd.len = strlen(passwd.arr);
EXEC SQL CONNECT :user IDENTIFIED BY :passwd;

/* Eingabe */
i = 0;
printf ("%s%s", "Bitte geben Sie eine Produktbezeichnung oder \"ende\" ein.\n");
scanf ("%s", &(prod_bez[i]));
while ((i < MAXPRODUKTE) && (strcmp (prod_bez[i], "ende") != 0))
    {
    i++;
    printf ("%s%s", "Bitte geben Sie eine Produktbezeichnung oder \"ende\" ein.\n");
    scanf ("%s", &(prod_bez[i]));
    }; /*while*/
num_of_prod = i;
if (num_of_prod = = 0) exit(-1);
```

```
/* Aufbau der SQL-Anfrage */
strcpy (sqltext.arr, "SELECT Monat, Tag, Bestellungen.PNr, Bez, Menge ");
strcat (sqltext.arr, "FROM Bestellungen, Produkte ");
strcat (sqltext.arr, "WHERE Bestellungen.PNr = Produkte.PNr ");
strcat (sqltext.arr, "AND ( ");
strcat (sqltext.arr, "Bez LIKE \'");
strcat (sqltext.arr, prod_bez[0]);
strcat (sqltext.arr, "%\' ");
for (i=1; i < num_of_prod; i++) {
    strcat (sqltext.arr, "OR Bez LIKE \'");
    strcat (sqltext.arr, prod_bez[i]);
    strcat (sqltext.arr, "%\' ");
    }; /*for*/
strcat (sqltext.arr, ") ");
strcat (sqltext.arr, "ORDER BY Monat DESC, Tag DESC");
sqltext.len = strlen(sqltext.arr);

/*  Nur zu Testzwecken */
printf ("\nDie generierte SQL-Anfrage lautet:\n"); printf ("%s\n\n", sqltext.arr);
/*  zum Beispiel:
    SELECT Monat, Tag, Bestellungen.PNr, Bez, Menge
    FROM Bestellungen, Produkte WHERE Bestellungen.PNr = Produkte.PNR
    AND ( Bez LIKE 'Druck%' OR Bez LIKE 'Papier%' OR Bez LIKE 'Platte%' )
    ORDER BY Monat DESC, Tag DESC                                          */

EXEC SQL PREPARE DynQuery FROM :sqltext;
EXEC SQL DECLARE BestIterator CURSOR FOR DynQuery;
EXEC SQL OPEN BestIterator;
found = TRUE;
while (found)
    {
    EXEC SQL FETCH BestIterator INTO :monat, :tag, :pnr, :bez, :menge;
    if (sqlca.sqlcode = = 0) {
        bez.arr[bez.len] = '\0';
        /* Ausgabe */
        printf ("%d.%d.: %d Stueck %s (PNr %d)\n", tag, monat, menge, bez.arr, pnr);
        }
    else
        found = FALSE;
    }; /*while*/
EXEC SQL CLOSE BestIterator;
EXEC SQL COMMIT WORK RELEASE;
exit (0);

/* Fehlerbehandlung */
handle_error:
    printf ("\n*** Error %d in program. ***\n", sqlca.sqlcode);
    printf ("%.70s\n", sqlca.sqlerrm.sqlerrmc);
    exit (-1);
} /*main*/
```

Beispiele für Anwendungen, die dynamisches ESQL erfordern:

- SQL*Plus und ähnliche universelle, interaktive DBS-Schnittstellen sind Programme, die dynamisches ESQL verwenden.

- Anwendungen, die sowohl Daten als auch Metadaten (Relationen des DB-Katalogs) verarbeiten, benötigen in vielen Fällen dynamisches ESQL.
  Solche Anwendungen sind mit dynamischem ESQL realisierbar, weil in praktisch allen relationalen DBS der DB-Katalog in Form von Relationen zugreifbar ist.

  Beispiel (Achtung: in SQL so nicht möglich !):
  SELECT * FROM * WHERE * LIKE '%Lafontaine%'

Wichtige Katalogrelationen in Oracle:

SYSCATALOG bzw. SYS.ALL_TABLES UNION SYS.ALL_VIEWS

OWNER	TABLE_NAME	TABLE_TYPE
DBS	BESTELLUNGEN	TABLE
DBS	KUNDEN	TABLE
DBS	PRODUKTE	TABLE
SYS	ALL_TABLES	VIEW
.		
.		
.		

SYS.ALL_TAB_COLUMNS

OWNER	TABLE_NAME	COLUMN_ NAME	DATA_ TYPE	DATA_ LENGTH	...	COLUMN_ ID	...
DBS	KUNDEN	KNR	NUMBER	22		1	
DBS	KUNDEN	NAME	CHAR	30		2	
DBS	KUNDEN	STADT	CHAR	30		3	
DBS	KUNDEN	SALDO	NUMBER	22		4	
DBS	KUNDEN	RABATT	NUMBER	22		5	
DBS	PRODUKTE	PNR	NUMBER	22		1	
.							
.							
.							
SYS	ALL_TABLES	OWNER	CHAR	30		1	
SYS	ALL_TABLES	TABLE_NAME	CHAR	30		2	
.							
.							
.							

Stored Procedures: Einbettung von SQL in eine 4GL
(Beispiel: PL/SQL von Oracle)

Die Prinzipien sind wie bei ESQL. Syntaktisch ergeben sich einige Vereinfachungen. Andererseits sind 4GLs häufig gegenüber Standardprogrammiersprachen wie (z.B. C) in ihren Datentypen und sonstigen Ausdrucksmitteln beschränkt.

Grobsyntax einer Prozedurdeklaration:

```
proc-decl =    CREATE PROCEDURE proc-name
               [ ”(" param-name type { ”," param-name type } ”)" ] AS proc-body
proc-body =    decl-part block
block =  BEGIN statement-list [ exception-part ] END
```

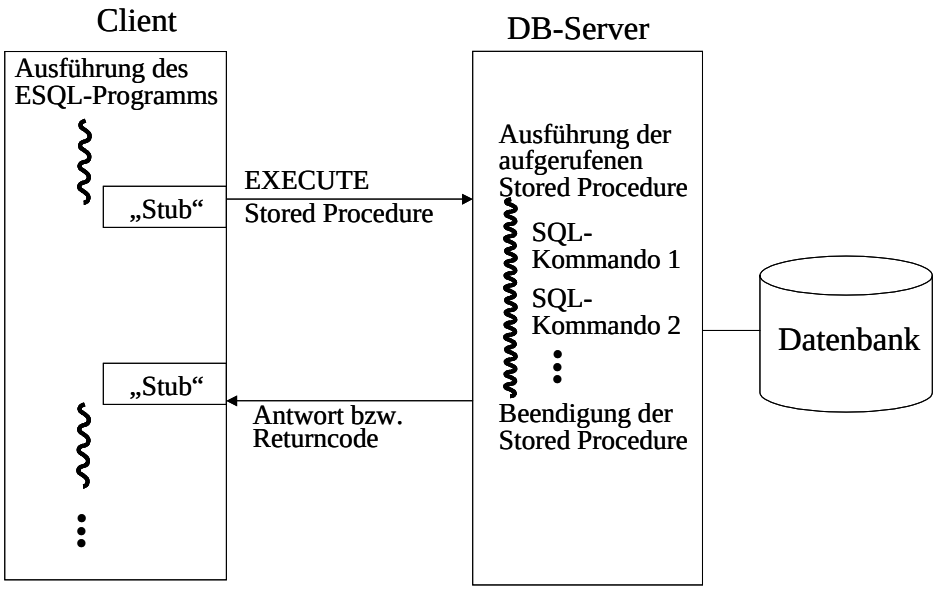
Beispiel:

```
CREATE PROCEDURE Lager_Auffuellen AS
DECLARE ...;
BEGIN
   DECLARE CURSOR Knappe_Produkte IS
      SELECT PNr, Vorrat
      FROM Produkte
      WHERE Vorrat < 100;
   KP Knappe_Produkte%ROWTYPE;
   BEGIN
      OPEN Knappe_Produkte;
      LOOP
         FETCH Knappe_Produkte INTO KP;
         EXIT WHEN Knappe_Produkte%NOTFOUND;
         ...
         IF KP.Vorrat > 10
         THEN
            EXECUTE Nachbestellen (KP.PNr, 1000 - KP.Vorrat)
         ELSE
            EXECUTE Express_Bestellen (KP.PNr, 10);
            EXECUTE Nachbestellen (KP.PNr, 1000 - KP.Vorrat - 10);
         END IF;
      END LOOP;
   END;
END;
```

Aufruf der PL/SQL Stored Procedure direkt von SQL*PLus, SQL*Forms oder ESQL-Programmen möglich,
z.B.: EXECUTE Lager_Auffuellen

Mehrere Prozeduren und Funktionen können zu einem Modul - in Oracle-Terminologie einem *Package* - zusammengefaßt und unter einem global bekannten Namen den Anwendungsentwicklern zur Verfügung gestellt werden.

Laufzeitarchitektur für Stored Procedures:



8.3 JDBC

Von Java-Programmen aus warden SQL-Datenbanken mit dem JDBC-Protokoll (Java Database Connectivity) angesprochen. Dies ist eine Java-orientierte konkrete Syntax für dynamisches ESQL (siehe 8.2). Details zu JDBC können in Handbüchern bzw. Dokumentations-Webseiten nachgelesen werden. Das folgende Beispielprogramm verdeutlicht die Programmierkonventionen von JDBC; es bezieht sich auf eine Datenbank mit den folgenden Relationen:

```
-- Kunden-Tabelle:
create table customer
   (name varchar(20) primary key, ipnumber varchar(20), city varchar(20));
-- Konto-Tabelle:
create table account (name varchar(20) primary key, balance integer);
-- Kunden-Tabelle:
create table history (name varchar(20), bookingdate date, amount integer);
```

Beispielprogramm

```
import java.io.*; import java.util.*; import java.sql.*;
public class myjdbc {
public static void main (String args[]) {

Connection con = null; Statement stmt = null;
String sqlstring;

String driver = "oracle.jdbc.driver.OracleDriver";
try { Class.forName(driver);}
  catch(Exception e) {
  System.out.println ("error when loading jdbc driver");};

try {
String jdbcURL = "jdbc:oracle:thin:";
String db = "(DESCRIPTION =(ADDRESS_LIST = (ADDRESS = " +
  "(PROTOCOL = TCP)(HOST = TOKYO)(PORT = 1521)))" +
  "(CONNECT_DATA =(SERVICE_NAME = TOKYO.WORLD)))";
String user = "lehre40"; String password = "leere4zig";
```

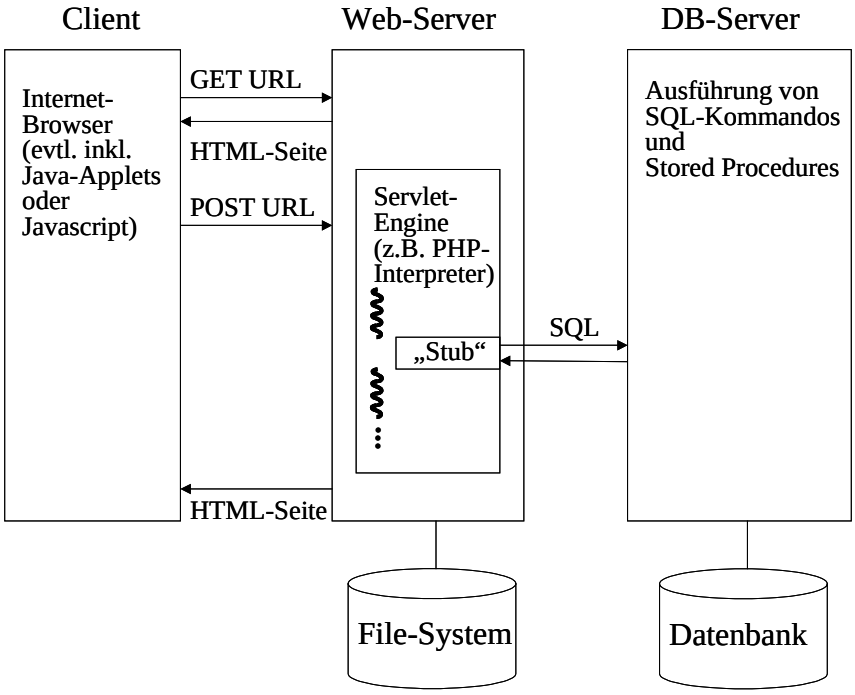
```
con = DriverManager.getConnection
     (jdbcURL + "@" + db, user, password);
con.setAutoCommit (false);
stmt = con.createStatement(); }
  catch (Exception e) {
  System.out.println ("error with jdbc connection"); };

String inputline = null;
try{
System.out.println ("\n Please type name. \n");
DataInputStream myinput = new DataInputStream (System.in);
inputline = myinput.readLine(); }
catch (IOException e) {System.out.println ("IO error");};

try{
sqlstring = "SELECT * FROM CUSTOMER " +
          "WHERE NAME LIKE '" + inputline + "'";
ResultSet result = stmt.executeQuery (sqlstring);
System.out.println ("CUSTOMER:");
System.out.println ("=========");
while (result.next()) {
  String namevar = result.getString("NAME");
  String ipnumbervar = result.getString("IPNUMBER");
  String cityvar = result.getString("CITY");
  System.out.println ("NAME: " + namevar + " IPNUMBER: "
                + ipnumbervar + " CITY: " + cityvar);
}; //while
con.commit(); }
  catch (SQLException sqlex)
  {System.out.println (sqlex.getMessage());};
System.out.println ("\n");
try {
sqlstring = "SELECT * FROM ACCOUNT " +
          "WHERE NAME LIKE '" + inputline +"'";
ResultSet result = stmt.executeQuery (sqlstring);

System.out.println ("ACCOUNT:");
System.out.println ("========");
while (result.next()) {
  String namevar = result.getString("NAME");
  int balancevar = result.getInt("BALANCE");
  System.out.println ("NAME: " + namevar +
                " BALANCE: " + balancevar);
}; //while
con.commit(); }
  catch (SQLException sqlex)
  {System.out.println (sqlex.getMessage());};

try{ con.close(); }
  catch (SQLException sqlex)
  {System.out.println (sqlex.getMessage());};

}; //main
} //myjdbc
```

8.4 Web-Anwendungen mit Java Servlets

Web-fähige Datenbankanwendungen basieren in der Regel auf Servlets, relativ kleinen Programmen bzw. Skripts, die unter der Kontrolle eines Web-Servers bzw. einer Servlet-Engine laufen (z.B. Apache/Tomcat). Servlets werden durch eine HTTP-Nachricht (GET oder POST) vom Web-Server aktiviert und generieren - typischerweise mit Hilfe von Datenbankzugriffen - dynamische HTML-Seiten, die der Web-Server an den Internet-Browser des Clients sendet.

Die syntaktisch an C und Perl angelehnte Skriptsprache PHP (PHP Hypertext Preprocessor) ist eine - vermutlich die populärste - Sprache zur Erstellung solcher Servlets. PHP wird direkt in eine HTML-Datei eingebettet, die die Datei-Extension .php haben muß. Wenn auf diese Datei via HTTP GET zugegriffen wird, interpretiert der Web-Server die Datei zunächst als HTML-Seite; er erkennt eingebetteten PHP-Code durch ein Tag <?php ... ?> und schaltet die eigentliche PHP-Engine zur Interpretation des PHP-Codes ein. Die - mittels echo oder printf erzeugte - Ausgabe des PHP-Codes wird einfach an die entsprechenden Stellen der HTML-Seite hineinkopiert, und die so erzeugte dynamische HTML-Seite wird an den Internet-Browser geschickt.

Laufzeitarchitektur für Web-fähige Servlets



Web-Anwendungen, die mit Servlets arbeiten, haben eine 3-Schichten-Architektur (3-Tier Architecture) mit:

- einem Frontend-Client, typischerweise ein Internet-Browser, in dem ggf. Javascript-Code oder ganze Java-Applets laufen können,
- dem Applikations-Server in der Mitte, der Web-Server und Servlet-Engine umfasst, und
- einem Backend-Datenbank-Server, der vom Applikations-Server mit ODBC oder JDBC angesprochen wird.

Gegenüber einer 2-Schichten-Architektur, bei der ein mächtigerer Client direkt mit dem Datenbank-Server verbunden ware, hat die 3-Schichten-Architektur gewichtige Vorteile:

- Sie ist von der Code-Installation viel leichter zu verwalten und zu pflegen.
- Sie ist sicherer, da kein kritischer Code in den anfälligeren Clients läuft.
- Sie ist weitaus besser skalierbar (da sie z.B. weniger Datenbank-Sessions verwendet).

Servlets in Java arbeiten nach demselben Grundprinzip, nur dass der Java-Code nicht direkt in HTML-Seiten eingebettet ist; dazu müsste man sogenannte JSPs (Java Servlet Pages) verwen-

den. Zum einfacheren Umgang mit dem Input des zum Servlet-Start führenden HTTP-Aufrufs und der HTTP-Antwort verwenden Java-Servlets typischerweise die JSDK-Klasse HttpServlet, die sie selbst geeignet erweitern können.

Das folgende Beispiel für ein Java-Servlet arbeitet auf einer Datenbank mit den folgenden Relationen:

```
-- Kunden-Tabelle:
create table customer
   (name varchar(20) primary key, ipnumber varchar(20), city varchar(20));
-- Konto-Tabelle:
create table account (name varchar(20) primary key, balance integer);
-- Kunden-Tabelle:
create table history (name varchar(20), bookingdate date, amount integer);
```

Das Servlet arbeitet nach der folgenden Ablauflogik:

Empfange HTTP-Auftrag (Aufruf der Webseite);
Versuche Kunden aufgrund seiner IP-Nummer in der Datenbank zu finden;
Prüfe, ob der Kunde Parameter im Eingabeformular der Webseite angegeben hat;
if Kunde ist in Datenbank oder Kundendaten im Eingabeformular übergeben
then
Speichere Kundendaten in der Datenbank;
Erzeuge Text in HTTP-Antwort mit perönlicher Begrüßung;
else Sende leeres Eingabeformular;
fi;
Erzeuge vollständige Webseite als HTTP-Antwort mit Links auf Servlets „transfer“ und „audit“;
Sende HTTP-Antwort;

Beispiel-Servlet:

```
import java.lang.*; import java.io.*; import java.util.*;
import javax.servlet.*; import javax.servlet.http.*;
import java.sql.*; import java.net.*;
public class welcome extends HttpServlet {

public void doGet
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

String namevar; String cityvar;
BufferedReader readervar; String linestring;
String ipnumbervar = null; String browservar = null;
boolean newcustomer; String nexturl;

PrintWriter out = response.getWriter();
out.println("<html><body>");
namevar = request.getParameter ("inputname");
cityvar = request.getParameter ("inputcity");
ipnumbervar = request.getRemoteAddr();
// create JDBC connection
// ...



// actual application code
newcustomer = true;
try{
```

```
sqlstring = "SELECT * FROM CUSTOMER " +
          "WHERE IPNUMBER='" + ipnumbervar + "'";
ResultSet result = stmt.executeQuery (sqlstring);
if (result.next()) {
   newcustomer = false;
   namevar = result.getString("NAME");
   cityvar = result.getString("CITY");
}; //if
}
catch (SQLException sqlex)
   {System.out.println (sqlex.getMessage());};

if ((namevar != "") & (namevar != null)) {
  // insert customer info into db
  if (newcustomer){
    try {
    sqlstring = "INSERT INTO CUSTOMER " +
        "(NAME,IPNUMBER,CITY) VALUES (' " + namevar +
        "','" + ipnumbervar + "','" + cityvar + "')";
    stmt.execute (sqlstring);
    sqlstring = "INSERT INTO ACCOUNT (NAME,BALANCE) " +
        " VALUES (' " + namevar + "',0)";
    stmt.execute (sqlstring); con.commit(); }
    catch (SQLException sqlex)
      {System.out.println (sqlex.getMessage());};
   } } //then
else {
  // post form for inquiring customer info
  out.println("<form method=post action=welcome>");
  out.println("<br> Please type your name and city. <br>");
  out.println("Name: <input type=text name=inputname><br>");
  out.println("City: <input type=text "  +
            "name=inputcity value=Saarbruecken><br>");
  out.println("<input type=submit name=inputenter " +
            "value=\"Here you go!\"><br>");
  out.println("</form>");
}; //if

if ((namevar != "") && (namevar != null)) {
  out.println ("Welcome, " + namevar + "!<br>");
  out.println ("Today's date is ");
  out.println (new java.util.Date());
  out.println ("<br>");
  out.println ("Your IP number is " + ipnumbervar + "<br>");
  out.println ("How is the weather in " +
             cityvar + "?<br><br>");
  out.println ("How can we help you today?<br>");
  nexturl = "transfer?customername=" +
          URLEncoder.encode(namevar);
  out.println ("<a href=" + nexturl +
             "> Deposit or withdraw money</a><br>");
  nexturl = "audit?customername=" +
          URLEncoder.encode(namevar);
  out.println ("<a href=" + nexturl +
             "> Audit trail of your account</a><br>");
}; //if
out.println("</body></html>");
} //doGet
```

```
public void doPost
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

   doGet(request, response);

}  //  doPost

} // class welcome
```

In vielen Web-Anwendungen müssen ganze Sitzungen mit mehreren Dialogschritten zwischen Client und Applikations-Server verwaltet werden. Die Klasse HttpServlet bietet dazu Session-Objekte an, über die ein Client und der dazugehörige Session-Kontext vom Servlet eindeutig identifiziert werden können, obwohl das Servlet selbst in jedem Dialogschritt „quasi-zustandslos“ aufgerufen wird. Die Implementierung der Session-Objekte verwendet sog. Cookies, die daher im Browser aktiviert sein müssen. Cookies sind kurze Identifikatoren, die vom Web-Server erzeugt werden und zwischen Client und Server als Teil der HTTP-Header ständig hin- und hergeschickt werden. Für den Programmierer des Java-Servlets sind diese Details verborgen; er/sie muss lediglich im Servlet-Code für die Abfrage des Session-Objekts sorgen. Das folgende Beispielprogramm verdeutlicht diese – typische – Vorgehensweise.

Java-Servlet mit Sessions:

```
import java.lang.*; import java.io.*; import java.util.*;
import javax.servlet.*; import javax.servlet.http.*;
public class sessionexample extends HttpServlet {

public void doGet
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

Integer mycounter;
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h1>Welcome!</h1><br><br>");


// create session if not yet existing,
// otherwise fetch session data
HttpSession session = request.getSession(true);
if (session.isNew()) {
   mycounter = new Integer(1);
   session.putValue("mycounter", mycounter); }
else {
   mycounter = (Integer) session.getValue("mycounter");
   mycounter = new Integer(mycounter.intValue() + 1);
   out.println("You are visiting us the ");
   out.println(mycounter.intValue());
   out.println("th time");
   session.putValue("mycounter", mycounter);
}; //if
out.println("</body></html>");
} //doGet

} // class sessionexample
```

Ergänzende Literatur zu Kapitel 8:

J. Melton, A. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, 1993

Oracle9i Pro*C/C++ Precompiler Programmer's Guide

J.W. Schmidt, F. Matthes, The DBPL Project: Advances in Modular Database Programming, Information Systems Vol. 19 No. 2, 1994

G. Saake, K.-U. Sattler, Datenbanken und Java, dpunkt-Verlag, 2000

J. Hunter, Java Servlet Programming, O'Reilly, 2001

B.W. Perry, Java Servlet & JSP Cookbook, O'Reilly, 2003

A.Harbourne-Thomas et al., Professional Java Servlets 2.3, Wrox Press, 2002

Sun Microsystems, JDBC Technology, http://java.sun.com/products/jdbc/

Oracle9*i* JDBC Developer's Guide and Reference

Oracle9i JDBC Reference