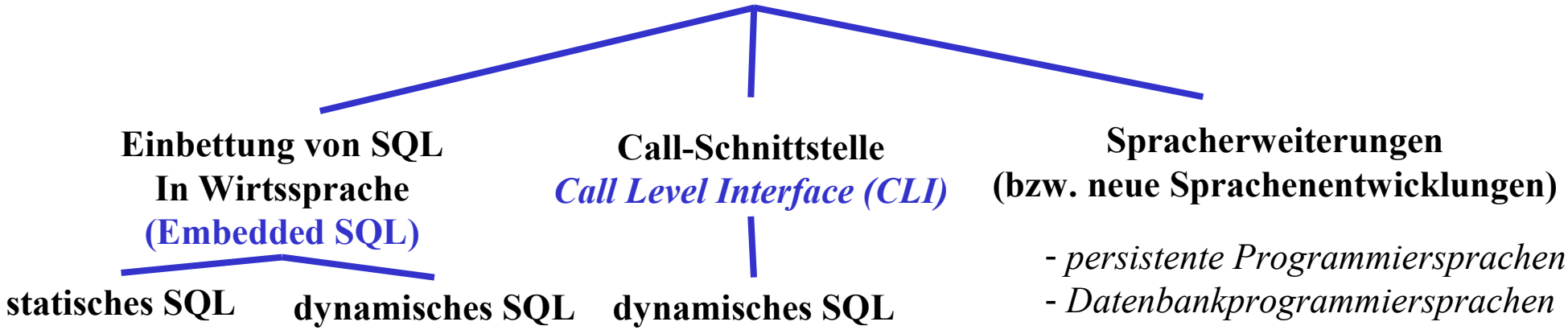
1. DB Anwendungsprogrammierung und Web-Anbindung

- Einleitung: Kopplung DBS - Wirtssprache
- Eingebettetes SQL
  - Cursor-Konzept
  - positionierte Änderungsoperationen (UPDATE, DELETE)
  - Dynamisches SQL
- Call-Level-Interface
- JDBC und SQLJ
- Gespeicherte Prozeduren (Stored Procedures)
  - Prozedurale Spracherweiterungen von SQL (PSM)
  - Gespeicherte Prozeduren mit Java
- Web-Anbindung
  - Architekturen Web-Informationssysteme
  - CGI, Servlets, JSP, PHP

Kopplung mit einer Wirtssprache



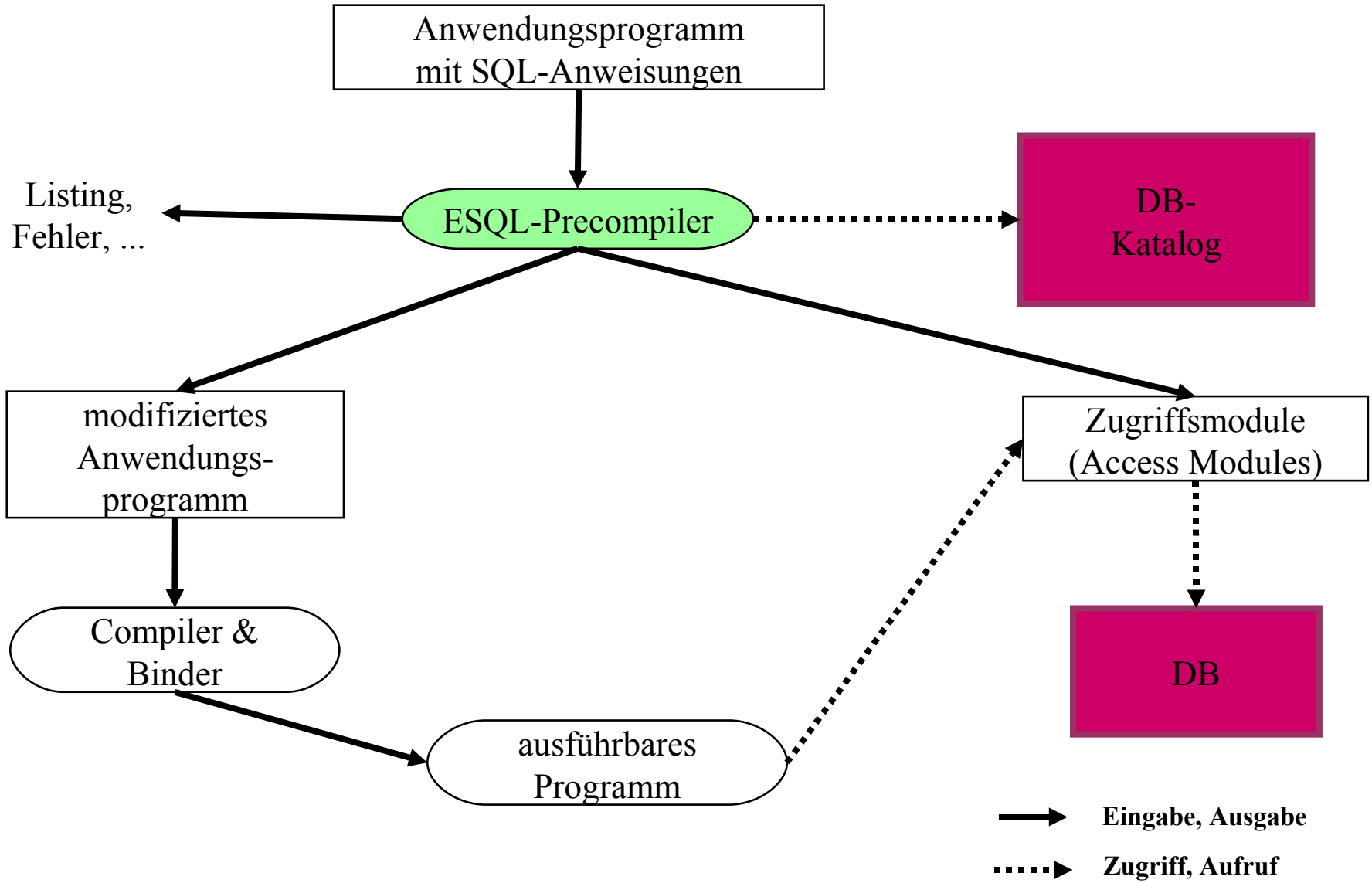
- Einbettung von SQL (Embedded SQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - Vorübersetzer (Prä-Compiler) wandelt DB-Aufrufe in Prozeduraufrufe um
- Call-Schnittstelle (CLI)
  - DB-Funktionen werden durch Bibliothek von Prozeduren realisiert
  - Anwendung enthält lediglich Prozeduraufrufe
  - weniger komfortable Programmierung als mit Embedded SQL
- Statisches SQL: Anweisungen müssen zur Übersetzungszeit feststehen
  - Optimierung zur Übersetzungszeit ermöglicht hohe Leistung
- Dynamisches SQL: Konstruktion von SQL-Anweisungen zur Laufzeit

Statisches SQL: Beispiel für C

```
exec sql include sqlca; /* SQL Communication Area */
main ()
{
exec sql begin declare section;
   char  X[8];
   int   GSum;
exec sql end declare section;
exec sql connect to dbname;
exec sql insert into PERS (PNR, PNAME) values (4711, 'Ernie');
exec sql insert into PERS (PNR, PNAME) values (4712, 'Bert');
printf("ANR ? "); scanf(" %s", X);
exec sql select sum (GEHALT) into :GSum from PERS where ANR = :X;
printf("Gehaltssumme: %d\n", GSum)
exec sql commit work;
exec sql disconnect;
}
```

- eingebettete SQL-Anweisungen werden durch "EXEC SQL" eingeleitet und spezielles Symbol (hier ";") beendet, um Compiler Unterscheidung von anderen Anweisungen zu ermöglichen
- Verwendung von AP-Variablen in SQL-Anweisungen verlangt Deklaration innerhalb eines "declare section"-Blocks sowie Angabe des Präfix ":" innerhalb von SQL-Anweisungen
- Werteabbildung mit Typanpassung durch INTO-Klausel bei SELECT
- Kommunikationsbereich SQLCA (Rückgabe von Statusanzeigern u. ä.)

Verarbeitung von ESQL-Programmen



Cursor-Konzept

- Kernproblem bei SQL-Einbettung in konventionelle Programmiersprachen: Abbildung von Tupelmengen auf die Variablen der Programmiersprache
- Cursor-Konzept zur satzweisen Abarbeitung von DBS-Ergebnismengen
  - Cursor ist ein Iterator, der einer Anfrage (Relation) zugeordnet wird und mit dessen Hilfe die Tupeln des Ergebnismenge einzeln (one tuple at a time) im Programm bereitgestellt werden
  - Trennung von Qualifikation (Query-Spezifikation) und Bereitstellung/Verarbeitung von Tupeln im Query-Ergebnis

Cursor-Konzept (2)

- Operationen auf einen Cursor C1
  - DECLARE C1 CURSOR FOR table-exp
  - OPEN C1
  - FETCH C1 INTO VAR1, VAR2, . . ., VARn
  - CLOSE C1
- Anbindung einer SQL-Anweisung an die Wirtssprachen-Umgebung
  - Übergabe der Werte eines Tupels mit Hilfe der INTO-Klausel bei FETCH
    => INTO target-commalist (Variablenliste d. Wirtsprogramms)
  - Anpassung der Datentypen (Konversion)
- kein Cursor erforderlich für Select-Anweisungen, die nur einen Ergebnissatz liefern (SELECT INTO)

Cursor-Konzept (3)

- Beispielprogramm in C (vereinfacht)

```
...
   exec sql begin declare section;
        char  X[50];
        char  Y[8];
        double G;
   exec sql end declare section;
   exec sql declare c1 cursor  for
            select NAME, GEHALT from PERS where ANR = :Y;
   printf("ANR ? "); scanf(" %s", Y);
   exec  sql  open  C1;
   while  (sqlcode == ok) {
           exec  sql  fetch  C1  into  :X, :G;
           printf("%s\n", X)}
   exec  sql  close  C1;
...
```

- DECLARE C1 . . . ordnet der Anfrage einen Cursor C1 zu
- OPEN C1 bindet die Werte der Eingabevariablen
- Systemvariable SQLCODE zur Übergabe von Fehlermeldungen (Teil von SQLCA)

Scroll-Cursor

```
DECLARE cursor [INSENSITIVE] [SCROLL] CURSOR FOR table-exp
          [ORDER BY order-item-commalist]
          [FOR {READ ONLY | UPDATE [OF column-commalist]}]
```

- Erweiterte Positionierungsmöglichkeiten durch SCROLL
  - Cursor-Definition (Bsp.):

```
EXEC SQL DECLARE  C2  SCROLL  CURSOR FOR
SELECT NAME, GEHALT FROM PERS
ORDER BY GEHALT ASCENDING
```

- Erweitertes FETCH-Statement:

```
EXEC SQL FETCH [[<fetch orientation>] FROM <cursor>  INTO  <target list>

Fetch orientation:  NEXT, PRIOR, FIRST, LAST,
                    ABSOLUTE  <expression>, RELATIVE <expression>
```

- Beispiele:

DB-Aktualisierung über Cursor

- Wenn die Tupeln, die ein Cursor verwaltet (active set), eindeutig Tupeln einer Relation entsprechen, können sie über Bezugnahme durch den Cursor geändert werden

```
positioned-update ::= UPDATE  table  SET  update-assignment-commalist
WHERE CURRENT OF  cursor

positioned-delete ::= DELETE  FROM  table WHERE CURRENT OF cursor
```

- Beispiel:

```
while  (sqlcode == ok) {
    exec sql fetch  C1  into  :X;
          /* Berechne das neue Gehalt in Z */
    exec sql update    PERS
                  set  GEHALT  =  :Z
                  where  current  of  C1;
  }
```

- keine Bezugnahme bei INSERT möglich!

Verwaltung von Verbindungen

- Zugriff auf DB erfordert i.a. zunächst, eine Verbindung herzustellen, v.a. in Client/Server-Umgebungen
  - Aufbau der Verbindung mit CONNECT, Abbau mit DISCONNECT
  - jeder Verbindung ist eine Session zugeordnet
  - Anwendung kann Verbindungen (Sessions) zu mehreren Datenbanken offenhalten
  - Umschalten der "aktiven" Verbindung durch SET CONNECTION

```
CONNECT TO   target [AS connect-name] [USER user-name]
```

```
SET  CONNECTION    { connect-name | DEFAULT }
```

```
DISCONNECT    { CURRENT | connect-name | ALL }
```

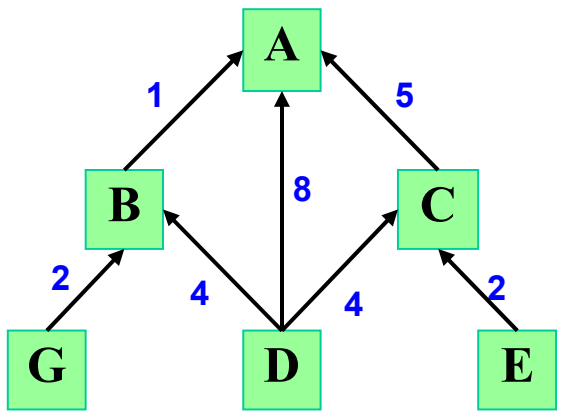
Beispiel: Stücklistenauflösung

- Darstellungsmöglichkeit im RM:

  TEIL (TNR, BEZ, MAT, BESTAND)

  STRUKTUR (OTNR, UTNR, ANZAHL)

Goes Into-Graph



TEIL

TNR	BEZ	MAT	BESTAND
A	Getriebe	-	10
B	Gehäuse	Alu	0
C	Welle	Stahl	100
D	Schraube	Stahl	200
E	Kugellager	Stahl	50
F	Scheibe	Blei	0
G	Schraube	Chrom	100

STRUKTUR

OTNR	UTNR	ANZAHL
A	B	1
A	C	5
A	D	8
B	D	4
B	G	2
C	D	4
C	E	2

- Aufgabe: Ausgabe aller Endprodukte sowie deren Komponenten

Beispiel: Stücklistenauflösung (2)

- max. Schachtelungstiefe sei bekannt (hier: 2)

```
exec sql begin declare section; char T0[10], T1[10], T2[10]; int ANZ;
exec sql end declare section;
exec sql declare C0 cursor for select distinct OTNR from STRUKTUR S1
    where not exists select * from STRUKTUR S2 where S2.UTNR = S1.OTNR);
exec sql declare C1 cursor for
    select UTNR, ANZAHL from STRUKTUR where OTNR = :T0;
exec sql declare C2 cursor for
    select UTNR, ANZAHL from STRUKTUR where OTNR = :T1;
exec sql open C0;
while (1) {
    exec sql fetch C0 into :T0;
    if (sqlcode == notfound) break;
    printf ("%s\n", T0);
    exec sql open C1;
    while (2) {exec sql fetch C1 into :T1, :ANZ;
               if (sqlcode == notfound) break;
               printf ("  %s: %d\n",T1, ANZ);
               exec sql open C2;
               while (3) {   exec sql fetch C2 INTO :T2, :ANZ;
                             if (sqlcode == notfound) break;
                             printf ("       %s: %d\n",T2, ANZ);  }
               exec sql close C2;  }
    exec sql close C1;  } /* END WHILE */
exec sql close (C0);
```

Dynamisches SQL

- dynamisches SQL: Festlegung von SQL-Anweisungen zur Laufzeit
  -> Query-Optimierung i.a. erst zur Laufzeit möglich
- SQL-Anweisungen werden vom Compiler wie Zeichenketten behandelt
  - Dekaration DECLARE STATEMENT
  - Anweisungen enthalten SQL-Parameter (?) statt Programmvariablen
- 2 Varianten: Prepare-and-Execute bzw. Execute Immediate

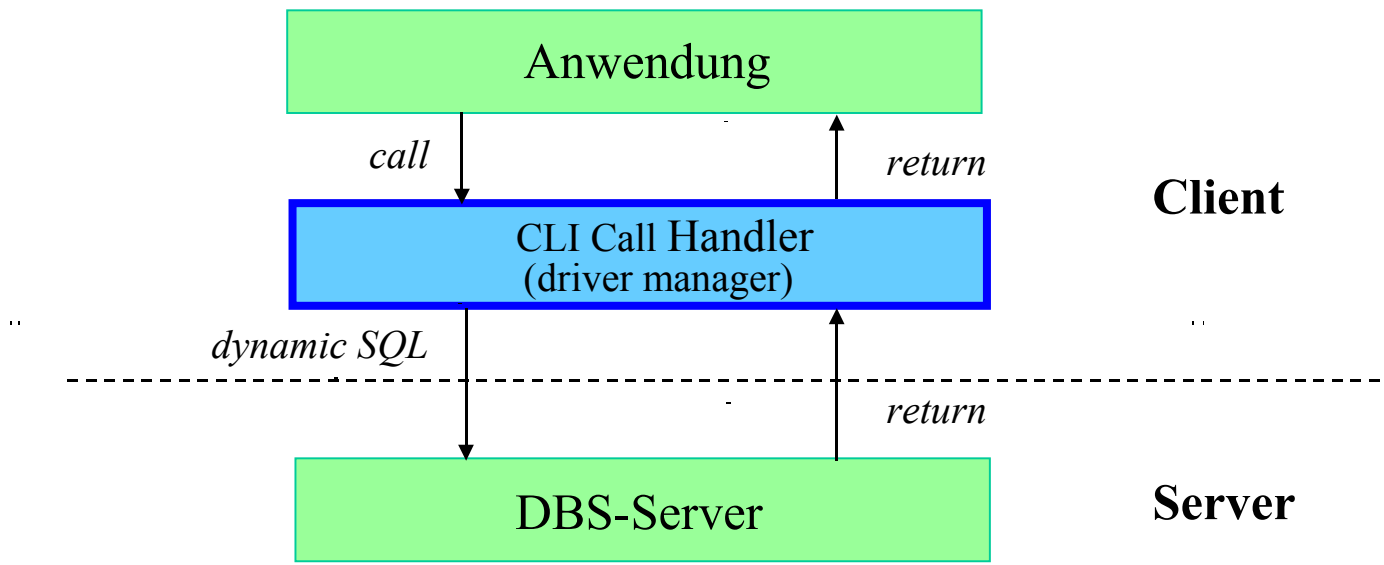
```
exec sql begin declare section;
   char  Anweisung[256], X[6];
exec sql end declare section;
exec sql declare SQLanw statement;
Anweisung = 'DELETE FROM PERS WHERE  ANR = ? AND ORT = ?'; /*bzw. Einlesen
exec sql prepare SQLanw from :Anweisung;
exec sql execute SQLanw using
scanf(" %s", X);
exec sql execute SQLanw using
```

- bei einmaliger Ausführung EXECUTE IMMEDIATE ausreichend

```
scanf(" %s", Anweisung);
exec sql execute immediate :Anweisung;
```

Call-Level-Interface

- alternative Möglichkeit zum Aufruf von SQL-Befehlen innerhalb von Anwendungsprogrammen: direkte Aufrufe von Prozeduren/Funktionen einer standardisierten Bibliothek (API)
- Hauptvorteil: keine Präkompilierung von Anwendungen
  - Anwendungen mit SQL-Aufrufen brauchen nicht im Source-Code bereitgestellt zu werden
  - wichtig zur Realisierung von kommerzieller Anwendungs-Software bzw. Tools
- Einsatz v. a. in Client/Server-Umgebungen



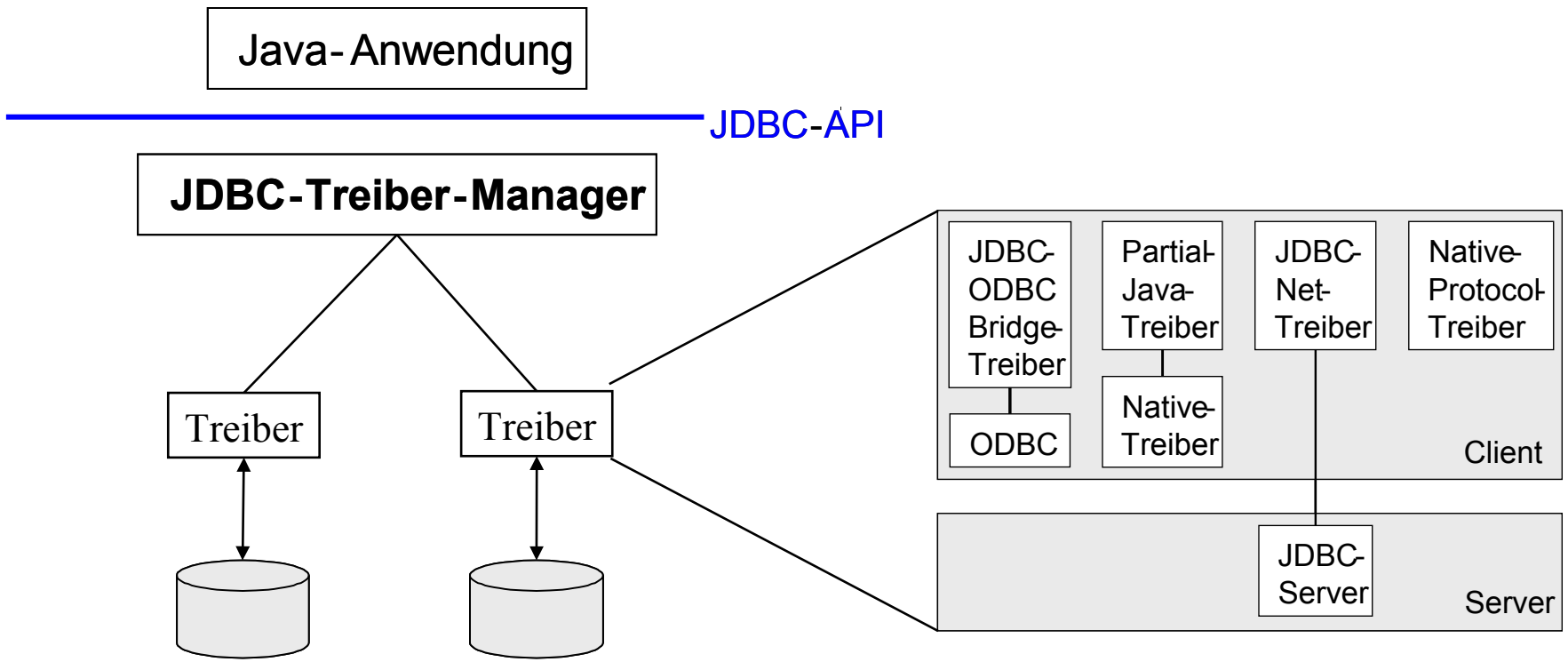
Call-Level-Interface (2)

- Unterschiede in der SQL-Programmierung zu eingebettetem SQL
  - CLI impliziert i.a. dynamisches SQL (Optimierung zur Laufzeit)
  - komplexere Programmierung
  - explizite Anweisungen zur Datenabbildung zwischen DBS und Programmvariablen
  - einheitliche Behandlung von mengenwertigen und einfachen Selects (<-> Cursor-Behandlung bei ESQL)

- SQL-Standardisierung des CLI erfolgte 1996
  - vorgezogener Teil von SQL99
  - starke Anlehnung an ODBC
  - über 40 Routinen: Verbindungskontrolle, Ressourcen-Allokation, Ausführung von SQL-Befehlen, Zugriff auf Diagnoseinformation, Transaktionsklammerung
- JDBC: neuere Variante

JDBC (Java Database Connectivity)*

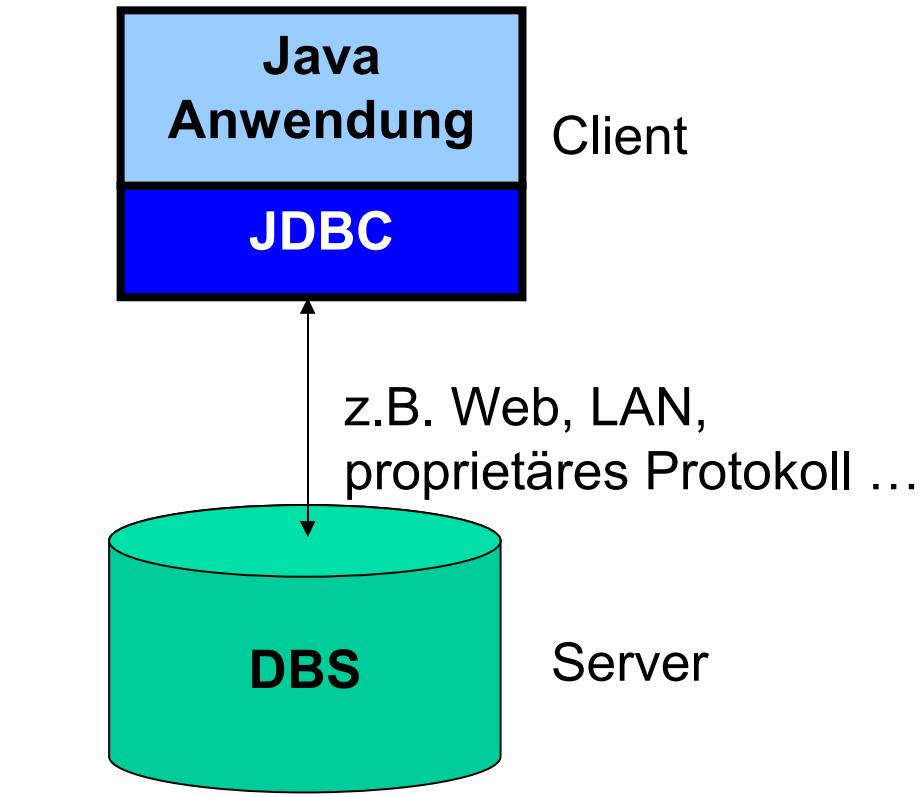
- Standardschnittstelle für den Zugriff auf SQL-Datenbanken unter Java
- basiert auf dem SQL/CLI (call-level-interface)
- Grobarchitektur



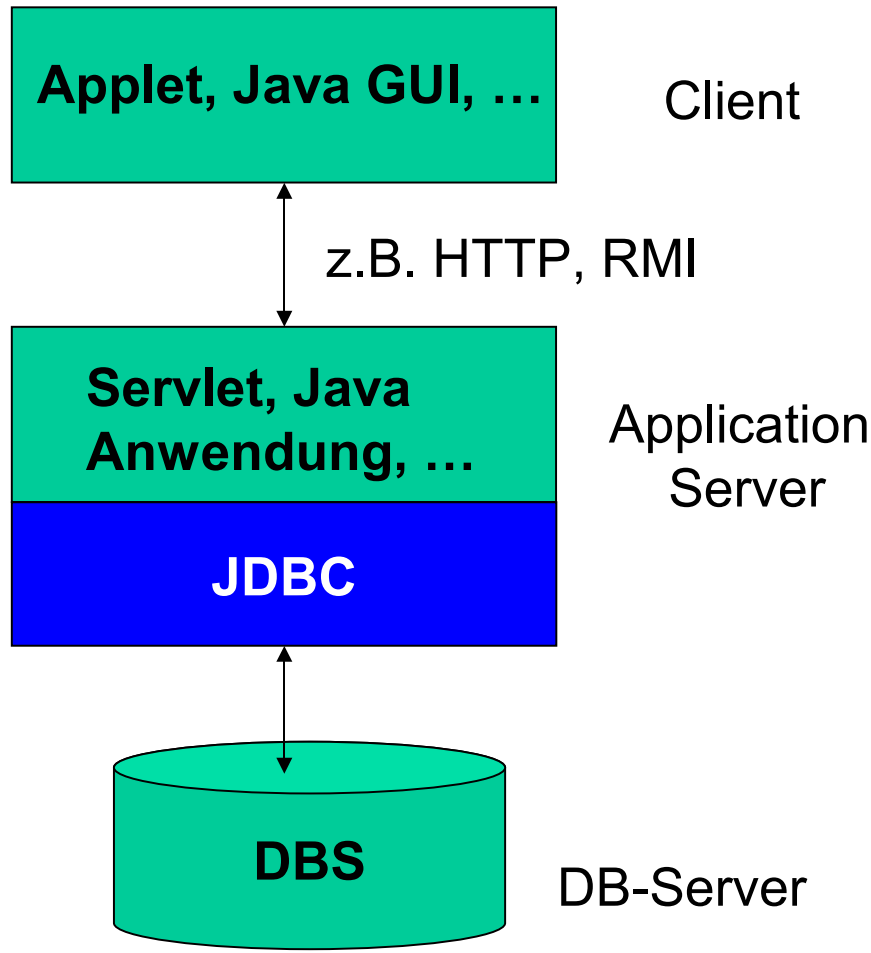
- durch Auswahl eines anderen JDBC-Treibers kann ein Java-Programm ohne Neuübersetzung auf ein anderes Datenbanksystem zugreifen

* http://java.sun.com/jdbc/

JDBC in Schichten-Architekturen



Zwei-Schichen-Architektur
(Two-Tier-Modell)



Drei-Schichten-Architektur
(Three-Tier-Modell)

JDBC: Grundlegende Vorgehensweise

- Schritt 1: Verbindung aufbauen

```
import java.sql.*;
        ...
Class.forName ( "COM.ibm.db2.jdbc.net.DB2Driver");
Connection con =
        DriverManager.getConnection ("jdbc:db2://host:6789/myDB", "login","pw");
```

- Schritt 2: Erzeugen eines SQL-Statement-Objekts

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

- Schritt 3: Statement-Ausführung

```
ResultSet rs = stmt.executeQuery ("SELECT matrikel FROM student");
```

- Schritt 4: Iterative Abarbeitung der Ergebnisdatensätze

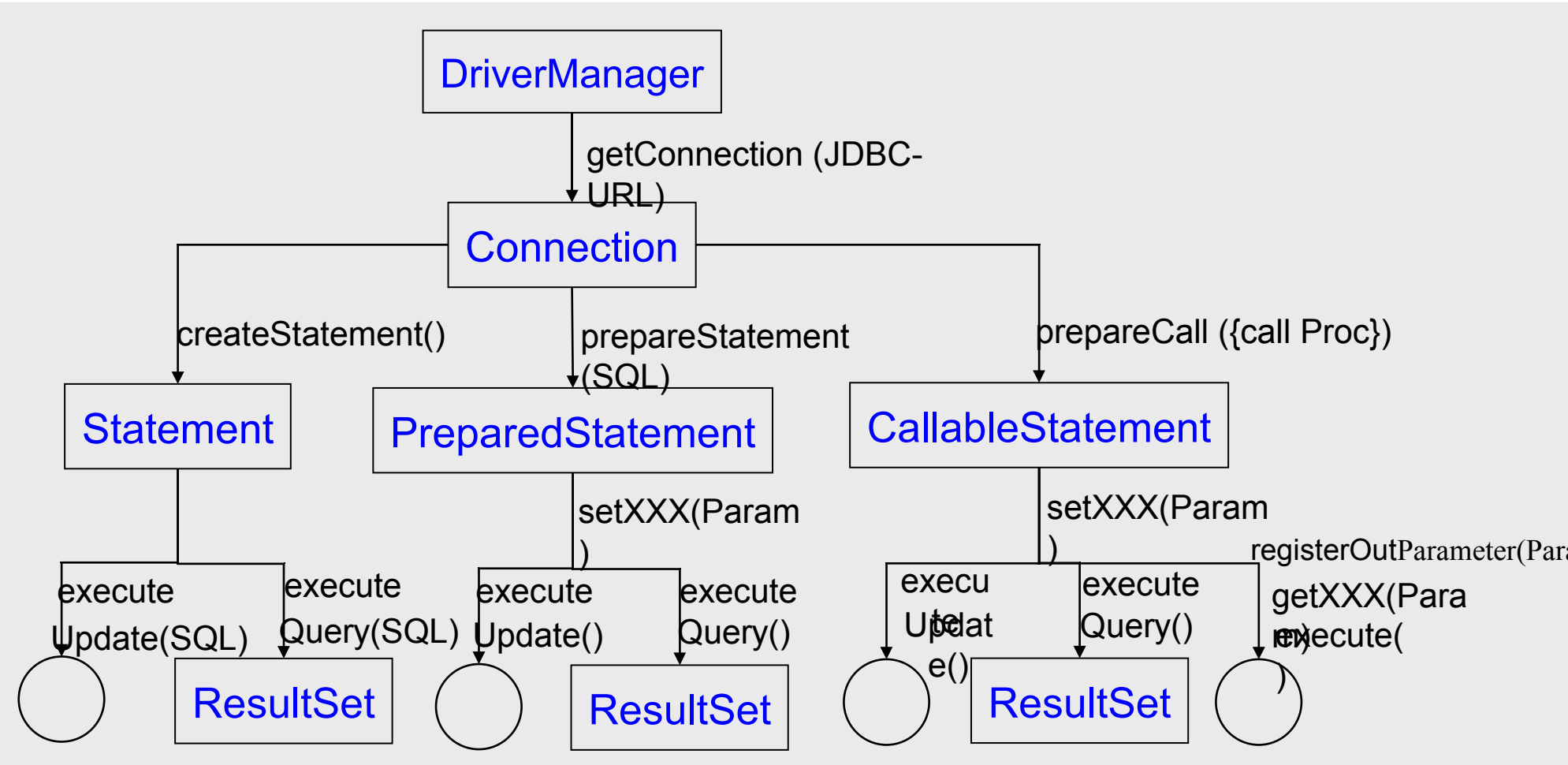
```
while (rs.next())
        System.out.println ("Matrikelnummer: " + rs.getString("matrikel"));
```

- Schritt 5: Schließen der Datenbankverbindung

```
con.close();
```

JDBC-Klassen

- streng typisierte objekt-orientierte API
- Aufrufbeziehungen (Ausschnitt)



JDBC: Beispiel

- Beispiel: Füge alle Matrikelnummern aus Tabelle ‘Student’ in eine Tabelle ‘Statistik’ ein

```
import java.sql.*;
 ...
public void copyStudents() {
 try {
   Class.forName („COM.ibm.db2.jdbc.net.DB2Driver“);// lade JDBC-Treiber (zur Laufzeit)
 } catch (ClassNotFoundException e) {// Fehlerbehandlung}
 try {
   String url = "jdbc:db2://host:6789/myDB"// spezifiziert JDBC-Treiber, Verbindungsdaten
   Connection con = DriverManager.getConnection(url, "login", "password");
   Statement stmt = con.createStatement();// Ausführen von Queries mit Statement-Objekt
   PreparedStatement pStmt = con.prepareStatement("INSERT INTO statistik (matrikel)
                                                   VALUES (?)");
                                 // Prepared-Stmts für wiederholte Ausführung

   ResultSet rs = stmt.executeQuery("SELECT matrikel FROM student");// führe Query aus

   while (rs.next()) {                    // lese die Ergebnisdatensätze aus
      String matrikel = rs.getString(1);  // lese aktuellen Ergebnisdatensatz
      pStmt.setString (1, matrikel);      // setze den Parameter der Insert-Anweisung
      pStmt.executeUpdate();              // führe Insert-Operation aus
   }
   con.close();
 } catch (SQLException e) { // Fehlerbehandlung}
}
```

JDBC: Transaktionskontrolle

- **Transaktionskontrolle** durch Methodenaufrufe der Klasse `Connection`
  - `setAutoCommit`: Ein-/Abschalten des Autocommit-Modus (jedes Statement ist eigene Transaktion)
  - `setReadOnly`: Festlegung ob lesende oder ändernde Transaktion
  - `setTransactionIsolation`: Festlegung der Synchronisationsanforderungen (None, Read Uncommitted, Read Committed, Repeatable Read, Serializable)
  - `commit` bzw. `rollback`: erfolgreiches Transaktionsende bzw. Transaktionsabbruch
- Beispiel

```
try {
   con.setAutoCommit (false);
   // einige Änderungsbefehle, z.B. Inserts
   con.commit ();
} catch (SQLException e) {
   try { con.rollback (); } catch (SQLException e2) {}
} finally {
   try { con.setAutoCommit (true); } catch (SQLException e3) {}
}
```

JDBC 2, JDBC 3

- Erweiterungen von JDBC 2
  - erweiterte Funktionalität von ResultSets (Scroll-Unterstützung; Änderbarkeit)
  - Batch Updates
  - Unterstützung von SQL99-Datentypen (BLOB, CLOB, ARRAY, REF, UDTs)
  - Optionales Package (javax.sql) mit Funktionen für Server-seitige Programme
    - JNDI (Java Naming and Directory Interface)-Unterstützung zur Spezifikation von Datenquellen
    - Connection Pooling
    - Verteilte Transaktionen (2-Phasen-Commit-Protokoll)
- Erweiterungen von JDBC 3
  - Package javax.sql ist nicht mehr optional
  - Unterstützung von Savepoints in Transaktionen
  - neue Datentypen BOOLEAN, DATALINK (Referenz auf externe Daten)
  - Zugriff auf Metadaten von UDTs und UDFs

SQLJ*

- Eingebettetes SQL (Embedded SQL) für Java
  - direkte Einbettung von SQL-Anweisungen in Java-Code
  - SQLJ-Programme müssen mittels Präprozessor in Java-Quelltext transformiert werden
- Spezifikation besteht aus
  - Embedded SQL für Java (Teil 0), *SQL Object Language Binding (OLB)*
  - Java Stored Procedures (Teil 1)
  - Java-Klassen für UDTs (Teil 2)
- Vorteile
  - Syntax- und Typprüfung zur Übersetzungszeit
  - Vor-Übersetzung (Performance)
  - einfacher/kompakter als JDBC
  - streng typisierte Iteratoren (Cursor-Konzept)

* http://www.sqlj.org

SQLJ (2)

- eingebettete SQL-Anweisungen: #sql [[<context>]] { <SQL-Anweisung> }
  - beginnen mit **#sql** und können mehrere Zeilen umfassen
  - können Variablen der Programmiersprache (:x) bzw. Ausdrücke (:y + :z) enthalten
  - können Default-Verbindung oder explizite Verbindung verwenden
- Vergleich SQLJ – JDBC (1-Tupel-Select)

SQLJ

```
#sql [con]{ SELECT name INTO :name
            FROM student WHERE matrikel = :mat};
```

JDBC

```
java.sql.PreparedStatement ps =
      con.prepareStatement („SELECT name “+
            „FROM student WHERE matrikel = ?“);
ps.setString (1, mat);
java.sql.ResultSet rs = ps.executeQuery();
rs.next()
name= rs.getString(1);
rs.close;
```

- Iteratoren zur Realisierung eines Cursor-Konzepts
  - benannte Iteratoren: Zugriff auf Spalten des Ergebnisses über Methode mit dem Spaltennamen
  - Positionsiteratoren: Zugriff über FETCH-Anweisung und Host-Variablen

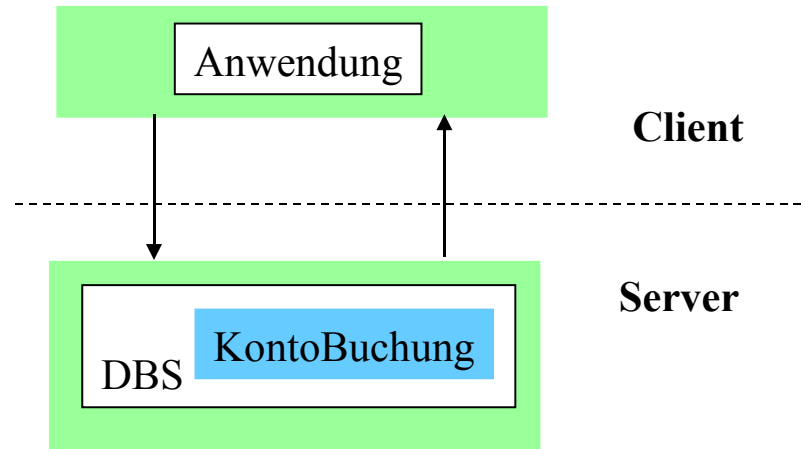
SQLJ (3)

- Beispiel: Füge alle Matrikelnummern aus Tabelle ‘Student’ in eine Tabelle ‘Statistik’ ein.

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
...
public void copyStudents() {
String drvClass= „COM.ibm.db2.jdbc.net.DB2Driver“;
   try {
    Class.forName(drvClass);
} catch (ClassNotFoundException e) { // errorlog }
try {
   String url = “jdbc:db2://host:6789/myDB”
   Connection con = DriverManager.getConnection
                          (url, “login”, “password”);
   // erzeuge einen Verbindungskontext
   // (ein Kontext pro Datenbankverbindung)
   DefaultContext ctx = new DefaultContext(con);
   // definiere Kontext als Standard-Kontext
   DefaultContext.setDefaultContext(ctx);
   // deklariere Typ für benannten Iterator
   #sql public iterator MatrikelIter (String matrikel);
   // erzeuge Iterator-Objekt
   MatrikelIter mIter;
   // Zuordnung und Aufruf der SQL-Anfrage
   #sql mIter = { SELECT matrikel FROM student };
   // navigiere über der Ergebnismenge
   while (mIter.next()) {
    // füge aktuelles Ergebnis in Tabelle Statistik ein
    #sql {INSERT INTO statistik (matrikel)
          VALUES ( mIter.matrikel()) };
    }
    mIter.close();
   } catch (SQLException e) { // errorlog }
}
```

Gespeicherte Prozeduren (Stored Procedures)

- Prozeduren, ähnlich Unterprogrammen einer prozeduralen Programmiersprache, werden durch DBS gespeichert und verwaltet
  - benutzerdefinierte Prozeduren oder Systemprozeduren
  - Programmierung der Prozeduren in SQL oder allgemeiner Programmiersprache



- Vorteile:
  - als gemeinsamer Code für verschiedene Anwendungsprogramme wiederverwendbar
  - Anzahl der Zugriffe des Anwendungsprogramms auf die DB werden reduziert
  - Performance-Vorteile v.a. in Client-Server-Umgebungen
  - höherer Grad der Isolation der Anwendung von DB wird erreicht
- Nachteile ?

Persistente SQL-Module (PSM)

- in SQL geschriebene Prozeduren erfordern Spracherweiterungen gegenüber SQL1992
  - u.a. allgemeine Kontrollanweisungen IF, WHILE, etc.
  - 1996: SQL-Standardisierung zu Persistent Storage Modules (PSM)
  - herstellerspezifische Festlegungen existieren bereits seit 1987 (Sybase Transact-SQL)
- PSM: vorab (1996) fertiggestellter Teil des SQL99-Standards
  - Routinen sind Schema-Objekte (wie Tabellen etc.) und können im Katalog aufgenommen werden (beim DBS/Server)
  - Routinen: Prozeduren und Funktionen
  - geschrieben in SQL (SQL routine) oder in externer Programmiersprache (external routine)
- zusätzliche DDL-Anweisungen
  - CREATE PROCEDURE, DROP PROCEDURE,
  - CREATE FUNCTION, DROP FUNCTION

PSM: Externe Routinen

- **externe Routinen** in beliebiger Programmiersprache (C, PASCAL, FORTRAN, ...)
  - Nutzung bestehender Bibliotheken
  - Akzeptanz
  - aber 2 Sprachen / Typsysteme
    - Typkonversion erforderlich
    - Typüberwachung außerhalb von SQL
- Beispiel einer externen Funktionsspezifikation:

```
DECLARE EXTERNAL        sinus (FLOAT)
  RETURNS FLOAT LANGUAGE FORTRAN;
```

PSM: SQL-Routinen

- **SQL-Routinen**: in SQL geschriebene Prozeduren/Funktionen
  - Deklarationen lokaler Variablen etc. innerhalb der Routinen
  - Nutzung zusätzlicher Kontrollanweisungen: Zuweisung, Blockbildung, IF, LOOP, etc.
  - Exception Handling (SIGNAL, RESIGNAL)
  - integrierte Programmierumgebung
  - keine Typkonversionen

- Beispiel

```
CREATE PROCEDURE KontoBuchung
    (IN konto INTEGER, IN betrag DECIMAL (15,2));
          BEGIN  DECLARE C1 CURSOR FOR ...;

                 UPDATE account
                 SET balance = balance + betrag
                 WHERE account_# = konto;
           ...
END;
```

- Prozeduren werden über CALL-Anweisung aufgerufen:

```
EXEC SQL CALL KontoBuchung  (:account_#, :balance);
```

PSM: SQL-Routinen (2)

- Beispiel einer SQL-Funktion:

```
CREATE FUNCTION vermoegen (kunr INTEGER)
                  RETURNS DECIMAL (15,2);

 BEGIN     DECLARE vm INTEGER;
           SELECT sum (balance) INTO vm
           FROM   account
           WHERE account_owner = kunr;
           RETURN vm;
 END;
```

- Aufruf persistenter Funktionen (SQL und externe) in SQL-Anweisungen wie Built-in-Funktionen

```
SELECT *
FROM kunde
WHERE vermoegen (KNR) > 100000.00
```

- Prozedur- und Funktionsaufrufe können rekursiv sein

Prozedurale Spracherweiterungen: Kontrollanweisungen

Compound Statement	BEGIN ... END;
SQL-Variablendeklaration	DECLARE var type;
If-Anweisung	IF condition THEN ... ELSE ... :
Case-Anweisung	CASE expression WHEN x THEN ... WHEN ... :
Loop-Anweisung	WHILE i < 100 LOOP ... END LOOP;
For-Anweisung	FOR result AS ... DO ... END FOR;
Leave-Anweisung	LEAVE ...;
Prozeduraufruf	CALL procedure_x (1, 2, 3);
Zuweisung	SET x = “abc”;
Return-Anweisung	RETURN x;
Signal/Resignal	SIGNAL division_by_zero;

PSM Beispiel

```
outer: BEGIN
   DECLARE account INTEGER DEFAULT 0;
   DECLARE balance DECIMAL (15,2);
   DECLARE no_money EXCEPTION FOR SQLSTATE VALUE 'xxxxx';
   DECLARE DB_inconsistent EXCEPTION FOR SQLSTATE VALUE 'yyyyy';

   SELECT account_#, balance INTO account, balance FROM accounts ...;
   IF (balance - 10) < 0 THEN SIGNAL no_money;
   BEGIN ATOMIC
     DECLARE cursor1 SCROLL CURSOR ...;
     DECLARE balance DECIMAL (15,2);
     SET balance = outer.balance - 10;
     UPDATE accounts SET balance = balance WHERE account_# = account;
     INSERT INTO account_history VALUES (account, CURRENT_DATE, 'W', balance); .....
   END;

  EXCEPTION
     WHEN no_money THEN
     CASE (SELECT account_type FROM accounts WHERE account_# = account)
           WHEN 'VIP' THEN INSERT INTO send_letter ....
           WHEN 'NON-VIP' THEN INSERT INTO blocked_accounts ...
     ELSE SIGNAL DB_inconsistent;

     WHEN DB_inconsistent THEN
           BEGIN .... END;

     END;
```

Gespeicherte Prozeduren in Java

- Erstellen einer Stored Procedure (z. B. als Java-Methode)

```
public static void kontoBuchung(int konto,
                        java.math.BigDecimal betrag,
                           java.math.BigDecimal[] kontostandNeu)
throws SQLException {
   Connection con = DriverManager.getConnection
                        ("jdbc:default:connection");
                              // Nutzung der aktuellen Verbindung
   PreparedStatement pStmt1 = con.prepareStatement(
          "UPDATE account SET balance = balance + ?
                WHERE account_# = ?");
   pStmt1.setBigDecimal( 1, betrag);
   pStmt1.setInt( 2, konto);
   pStmt1.executeUpdate();
   PreparedStatement pStmt2 = con.prepareStatement(
          "SELECT balance FROM account WHERE account_# = ?");
   pStmt2.setInt( 1, konto);
   ResultSet rs = pStmt2.executeQuery();
   if (rs.next()) kontostandNeu[0] = rs.getBigDecimal(1);
   pStmt1.close(); pStmt2.close(); con.close();
   return;
}
```

Gespeicherte Prozeduren in Java (2)

- Deklaration der Prozedur im Datenbanksystem mittels SQL

```
CREATE PROCEDURE KontoBuchung(   IN konto INTEGER,
                                 IN betrag DECIMAL (15,2),
                              OUT kontostandNeu DECIMAL (15,2))
 LANGUAGE java
 PARAMETER STYLE java
 EXTERNAL NAME 'myjar:KontoClass.kontoBuchung'
                 // Java-Archiv myjar enthält Methode
```

Gespeicherte Prozeduren in Java (3)

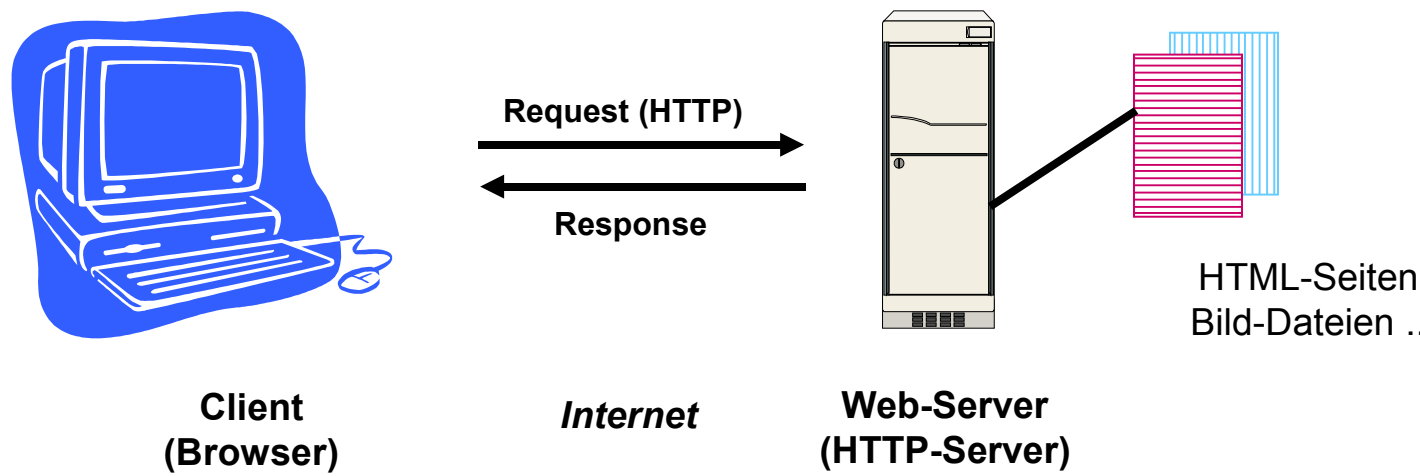
- Aufruf einer Stored Procedure in Java

```
public void ueberweisung(Connection con, int konto1, int konto2,
java.math.BigDecimal betrag)
throws SQLException {

  con.setAutoCommit (false);

  CallableStatement cStmt = con.prepareCall("{call KontoBuchung (?, ?, ?)}");
  cStmt.registerOutParameter(3, java.sql.Types.DECIMAL);

  cStmt.setInt(1, konto1);
  cStmt.setBigDecimal(2, betrag.negate());
  cStmt.executeUpdate();

  java.math.BigDecimal newBetrag = cStmt.getBigDecimal(3);
  System.out.println("Neuer Betrag: " + konto1 + " " + newBetrag.toString());

  cStmt.setInt(1, konto2);
  cStmt.setBigDecimal(2, betrag);
  cStmt.executeUpdate();

  newBetrag = cStmt.getBigDecimal(3);
  System.out.println("Neuer Betrag: " + konto2 + " " + newBetrag.toString());

  cStmt.close();
  con.commit ();

  return;
}
```

Architektur-Varianten für Web-Informationssysteme

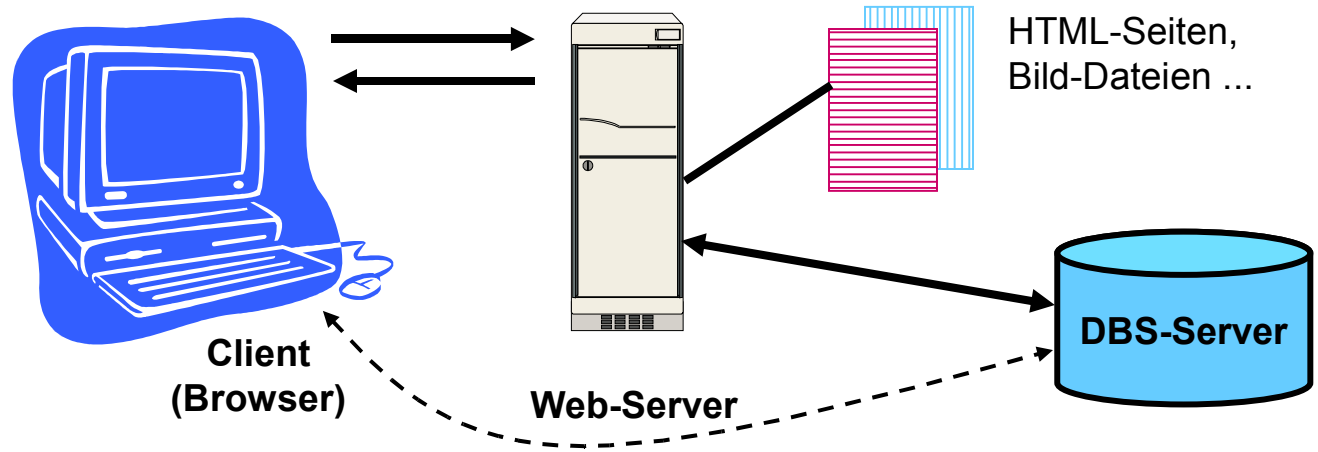
- statische Ansätze ohne DB
  - Daten liegen nur in Dateien
  - statisch festgelegter Informationsgehalt
  - relativ einfache Datenbereitstellung (HTML)
  - einfache Integration von Multimedia-Objekten (Bild, Video, ...) sowie externen Quellen
  - Aktualitätsprobleme für Daten und Links
  - oft Übertragung wenig relevanter Daten, hohe Datenvolumen . . .

Statische WebIS



Architektur-Varianten (2)

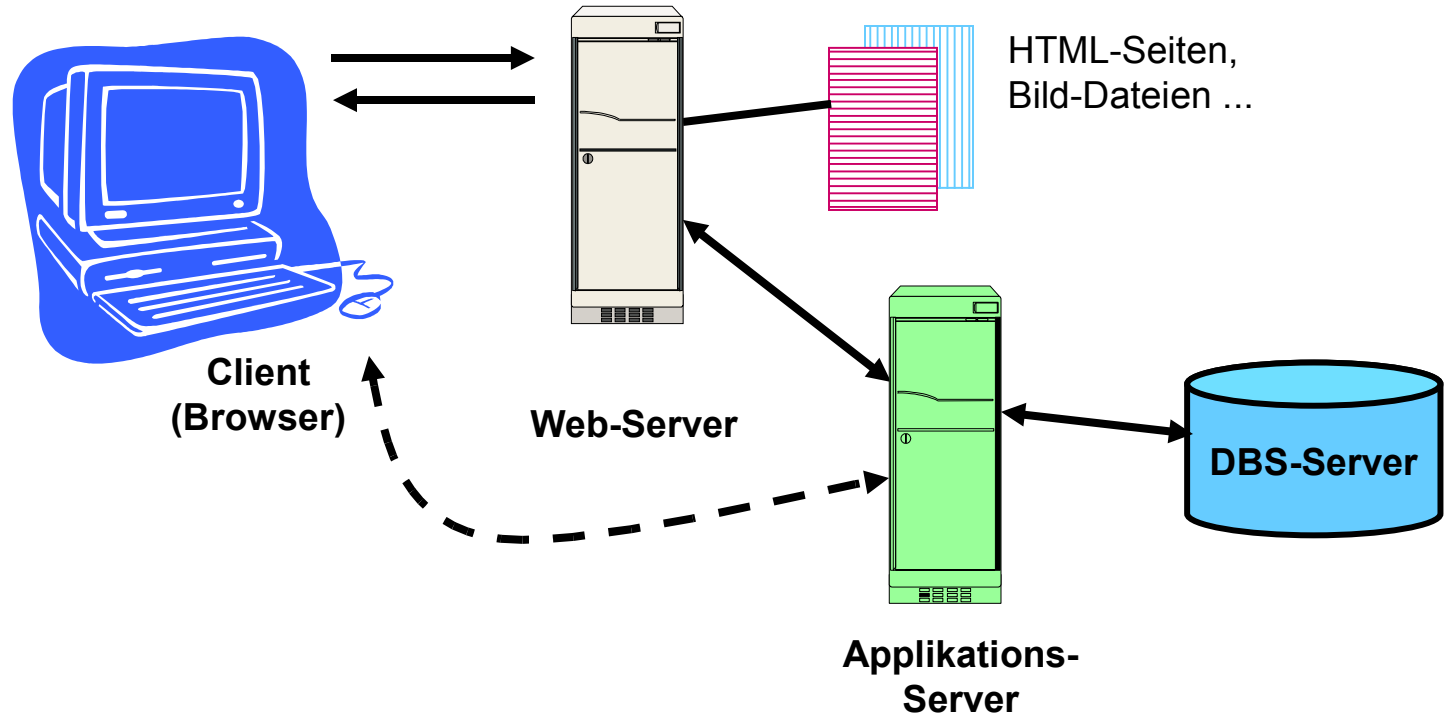
WebIS
mit
DB-Anbindung



- Anbindung von Datenbanksystemen
  - dynamischer Informationsgehalt durch Zugriff auf Datenbanken
  - Bereitstellung aktueller Informationen
  - bessere Skalierbarkeit
  - Transaktionsunterstützung / Mehrbenutzerfähigkeit (auch bei Änderungen) . . .
- Verwaltung/Speicherung von HTML/XML-Seiten, Dokumenten, Multimedia-Daten etc. durch DBS
  - Content Management / Website Management
  - hohe Flexibilität, bessere Konsistenzwahrung für Links ...

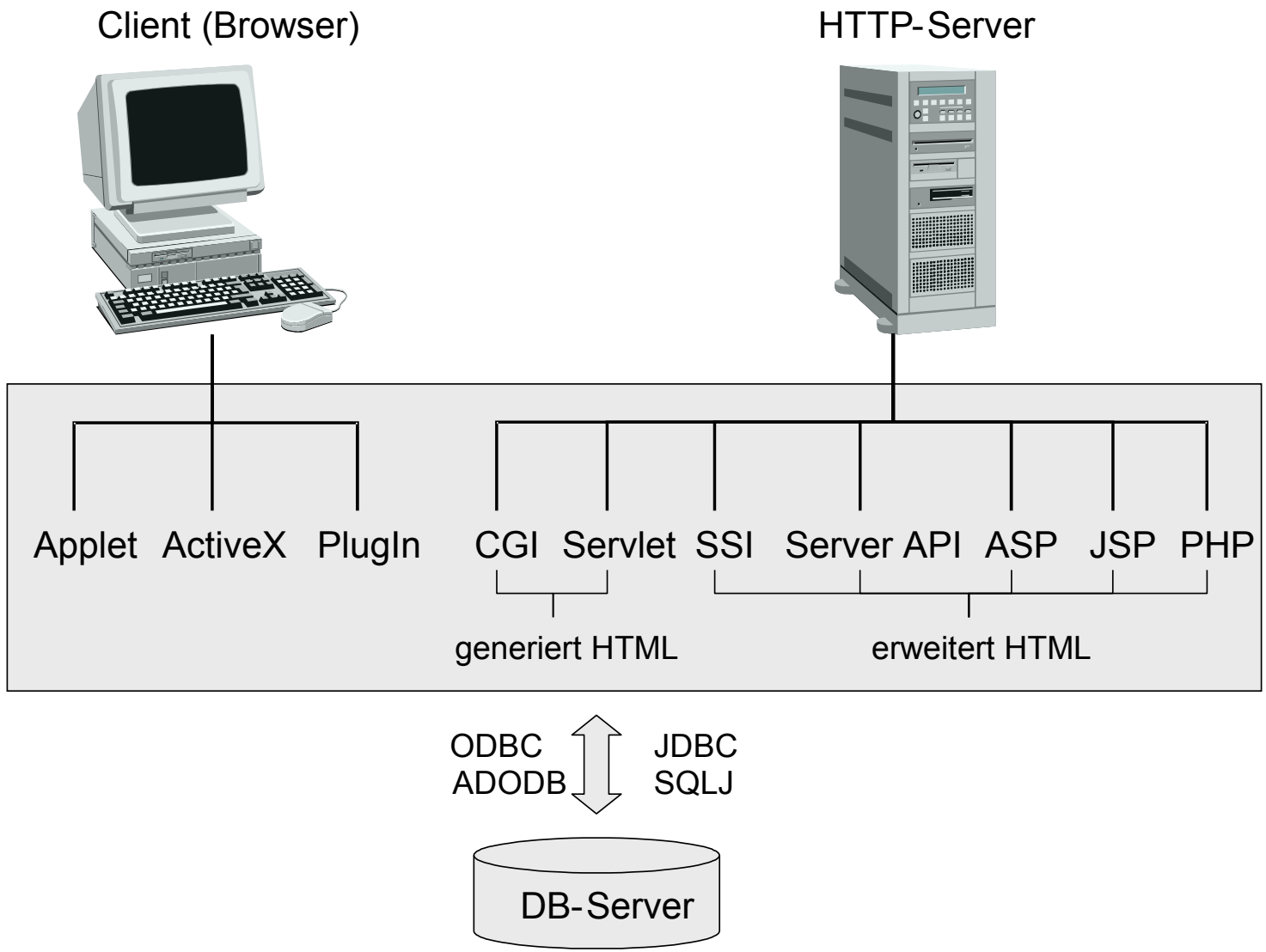
Architektur-Varianten (3)

Applikations-Orientierte WebIS



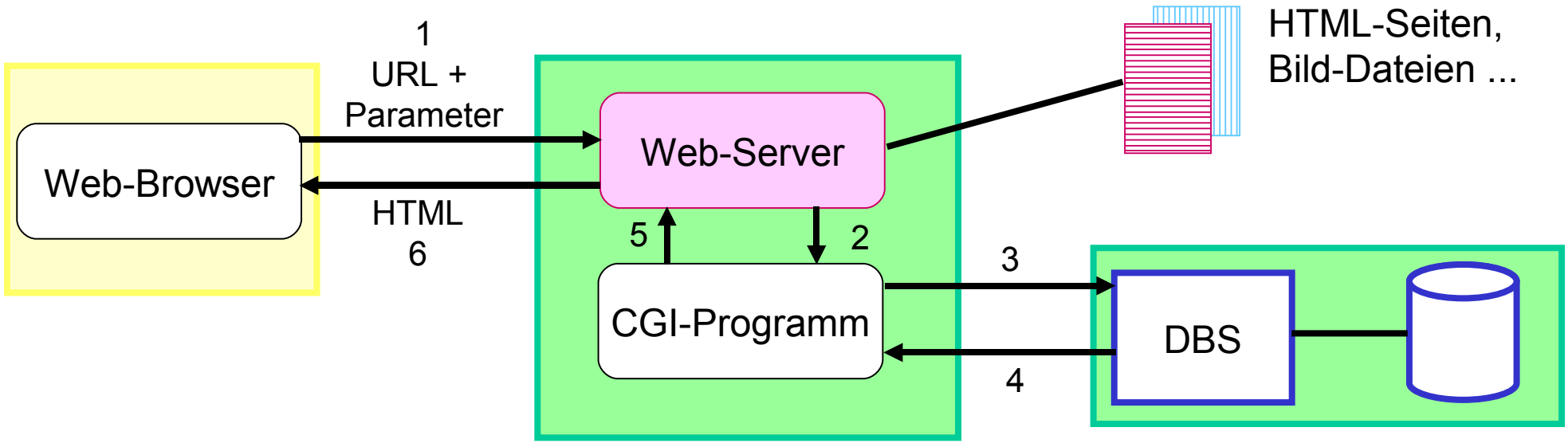
- Vorteile der Applikations-Server-Architektur
  - Unterstützung komplexer (Geschäfts-) Anwendungen
  - Skalierbarkeit (mehrere Applikations-Server)
  - Einbindung mehrerer DBS-Server / Datenquellen
  - Transaktions-Verwaltung
  - ggf. Unterstützung von Lastbalancierung, Caching, etc.

Übersicht Techniken zur Web-Anbindung von Datenbanken



Server-seitige Anbindung: CGI-Kopplung

- CGI: Common Gateway Interface
  - plattformunabhängige Schnittstelle zwischen Web-Server (HTTP-Server) und externen Anwendungen
  - wird von jedem Web-Server unterstützt



- CGI-Programme (z.B. realisiert in Perl, C, Shell-Skripte)
  - erhalten Benutzereingaben (aus HTML-Formularen) vom Web-Server als Parameter
  - können beliebige Berechnungen vornehmen und auf Datenbanken zugreifen
  - Ergebnisse werden als dynamisch erzeugte HTML-Seiten an Client geschickt

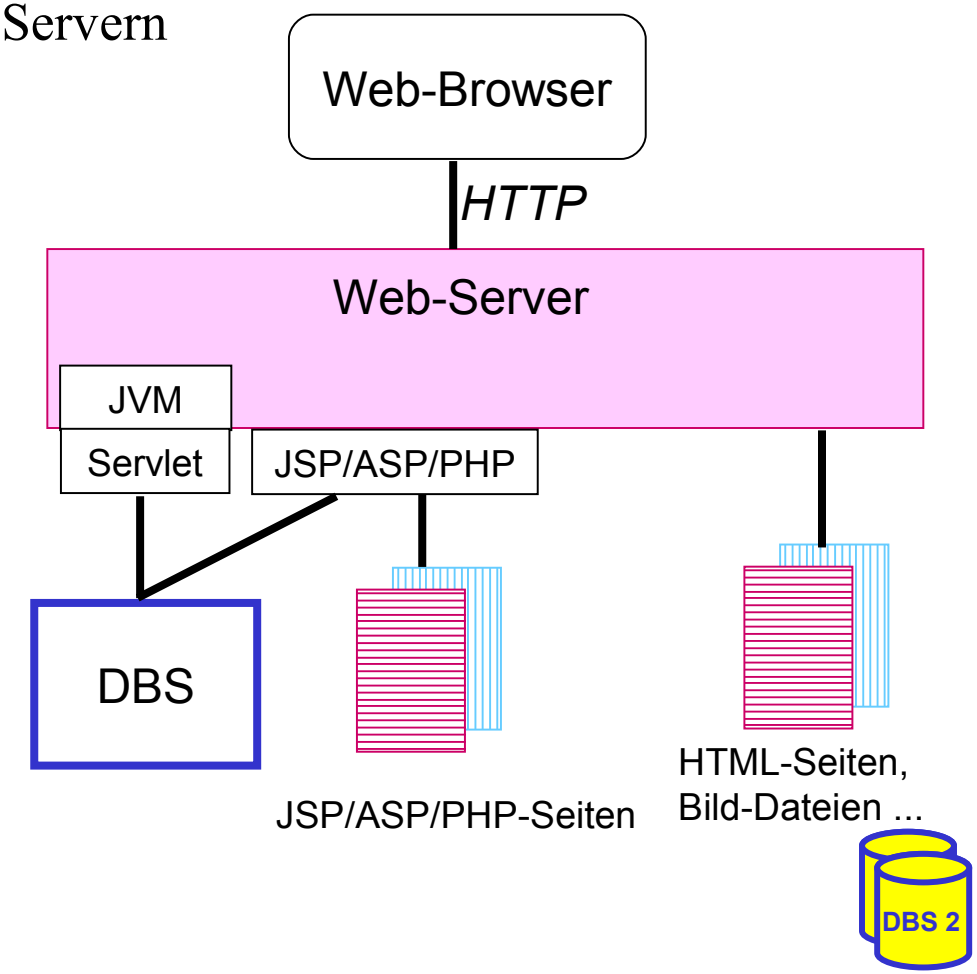
CGI-Kopplung (2)

- CGI-Programme generieren HTML-Ausgabe
- aufwendige / umständliche Programmierung
- mögliche Performance-Probleme
  - Eingabefehler werden erst im CGI-Programm erkannt
  - für jede Interaktion erneutes Starten des CGI-Programms
  - für jede Programmaktivierung erneuter Aufbau der DB-Verbindung

```
#!/bin/perl
use Msql;
# Seitenkopf ausgeben:
print"Content-type: text/html\n\n";
# [...]
# Verbindung mit dem DB-Server
herstellen:
$testdb = Msql->connect;
$testdb->selectdb("INFBIBLIOTHEK");
# DB-Anfrage
$q = $testdb->query
     ("select Autor, Titel  from ...");
# Resultat ausgeben:
print"<TABLE BORDER=1>\n"; print"<TR>\n
  <TH>Autor<TH>Titel</TR>";
  $rows = $q -> numrows;
while ($rows>0) {
      @sqlrow = $q->fetchrow;
      print      "<tr><td>",@sqlrow[0],
                 "</td><td>",
                 @sqlrow[1],</td></
tr>\n";
       $rows--; }
print"</TABLE>\n";
# Seitenende ausgeben
```

Server-seitige Web-Anbindung: weitere Ansätze

- Integration von CGI-Programmen in Web-Server
  - z.B. über proprietäre Web-Server-Lösungen wie ISAPI (Microsoft)
  - kein Starten eigener CGI-Prozesse, DB-Verbindungen können offen bleiben
- Einsatz von Java-Servlets
  - herstellerunabhängige Erweiterung von Web-Servern (Java Servlet-API)
  - Integration einer Java Virtual Machine (JVM) im Web-Server -> Servlet-Container
- server-seitige Erweiterung von HTML-Seiten um Skript-/Programmlogik
  - Java Server Pages
  - Active Server Pages (Microsoft-Lösung)
  - PHP-Anweisungen
- Integration von Auswertungslogik in DB-Prozeduren (stored procedures)



Servlets

- spezielle Java-Klassen, die in Servlet-Container ausgeführt werden
  - besondere Unterstützung von HTTP-Anfragen
  - Zugriff auf das gesamte Java-API, insbesondere auch auf JDBC-Klassen für DB-Zugriff
  - plattformunabhängig, serverunabhängig
- Servlet-Container für viele Web- und Application-Server verfügbar (Apache, IIS, IBM WebSphere, Tomcat ...)

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class EmpServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
 response.setContentType ("text/html");
 PrintWriter out = response.getWriter();
 out.println("<HTML><BODY>\n" +          "<H1>Employee #1</H1>\n");
try {// öffne Datenbankverbindung
 Connection con = DriverManager.getConnection( “jdbc:db2:myDB”, “login”, “password”);
       // erzeuge Statement-Objekt
 Statement stmt = con.createStatement();
```

Servlets (2)

```
ResultSet rs = stmt.executeQuery(“SELECT * FROM “ + „Employees WHERE EmployeeID = 1“;);
  if (rs.next()) {
  out.println(“<B>Name:</B>”+rs.getString(“Name”)+“<BR>”);
  out.println(“<B>Address:</B>+rs.getString(“Address”)+“<BR>”);
  out.println(“<B>City/State/ZIP:</B>”+rs.getString(“City”) + “ ”+rs.getString(“State”) +
                              “ ”+rs.getString(“ZipCode”));
        }
        rs.close();
        stmt.close();
        con.close();
  }
catch (SQLException e) { log(e.getMessage());
  }
  out.println (     "</BODY></HTML>");
}
```

Java Server Pages (JSP)

- Entwurf von dynamischen HTML-Seiten mittels HTML-Templates und XML-artiger Tags
  - Trennung Layout vs. Applikationslogik durch Verwendung von Java-Beans
  - Erweiterbar durch benutzerdefinierte Tags (z.B. für DB-Zugriff, Sprachlokalisierung, ...)
- JSP-Prozessor oft als Servlet realisiert
  - JSP-Seite wird durch JSP-Prozessor in ein Servlet übersetzt
  - JSP kann überall eingesetzt werden, wo ein Servlet-Container vorhanden ist

JSP-Seite:

```
<HTML>
<BODY>
 <jsp:useBean id="EmpData" class="FetchEmpDataBean,, scope="session">
 <jsp:setProperty name="EmpData“,, property="empNumber" value="1" />
 </jsp:useBean>
 <H1>Employee #1</H1>
 <B>Name:</B> <%=EmpData.getName()%><BR>
 <B>Address:</B> <%=EmpData.getAddress()%><BR>
 <B>City/State/Zip:</B>
 <%=EmpData.getCity()%>,
 <%=EmpData.getState()%>
 <%=EmpData.getZip()%>
</BODY>
</HTML>
```

JSP (2)

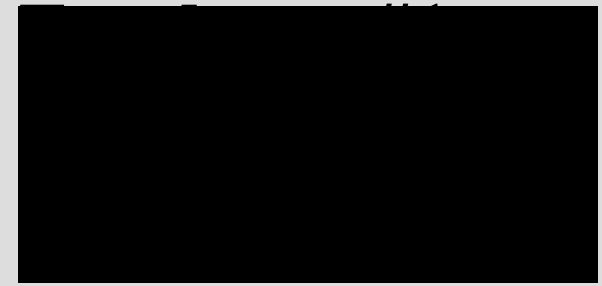
Bean:

```
class FetchEmpDataBean {
  private String name, address, city, state, zip;
  private int empNumber = -1;

  public void setEmpNumber(int nr) {
    empNumber = nr;
    try {
      Connection con = DriverManager.getConnection("jdbc:db2:myDB","login","pwd");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery ("SELECT * FROM Employees WHERE EmployeeID=" + nr);
      if (rs.next()) {
       name = rs.getString ("Name");
       address=rs.getString("Address");
       city = rs.getString ("City");
       state=rs.getString("State");
        zip=rs.getString("ZipCode");
      }
      rs.close();
     stmt.close();
     con.close();
    } catch (SQLException e) { //...}
  }
  public String getName() { return name; }
  public String getAddress() { return address; } ...
```

PHP (PHP: Hypertext Preprocessor)

- Open Source Skriptsprache zur Einbettung in HTML-Seiten
  - angelehnt an C, Java und Perl
  - besonders effizient bei der Erzeugung und Auswertung von HTML-Formularen
- erweiterbare Funktionsbibliothek (viele Module frei erhältlich)
  - Funktionen zum Datenbankzugriff sind DBS-spezifisch
  - DBS-unabhängiger Zugriff über ODBC oder Abstraktionsmodule (z. B. dbx, Pear::DB)

```
<html>
 <body>
  <h1>Employee #1</h1>
<?php
        $con = dbx_connect(DBX_PGSQL, "host", "myDB", "login", "password")
              or die("Verbindungsfehler!</body></html>");
        $result = dbx_query($con, "SELECT * FROM Employees WHERE EmployeeID = 1");
        if ( is_object($result) and ($result->rows > 0) ) {
?>
  <b>Name:</b> <?php echo $result->data[0]["Name"] ? > <br>
  <b>Address:</b>          <?php echo $result->data[0]["Address"] ?> <br>
  <b>City/State/ZIP:</b>  <?php echo $result->data[0]["City"].",".
        $result->data[0]["State"]."".$result->data[0]["ZipCode"]
              ?><br>
<?php   } else { echo "Unknown employee!"; } ?>
  </body>
</html>
```



Vergleiche

- Vergleich Servlets-JSP
  - beide sind Java-basiert / plattformunabhängig und können JDBC nutzen
  - Servlet: Layout (HTML) muß in Applikationslogik erzeugt werden; primär geeignet für Web-Seiten mit geringem Anteil an HTML-Code oder zur Erzeugung von Nicht-HTML-Daten
  - JSP: direkte Verwendung von HTML-Tags; Logik kann in Beans ausgelagert werden; getrennte Entwicklung von Web-Design und Datenverarbeitung (z.B. Datenbankschnittstelle) möglich

- Vergleich JSP-PHP
  - beides sind serverseitige Skriptsprachen zur Einbindung in HTML-Seiten
  - Seiten müssen gelesen und interpretiert werden
  - JSP-Vorteile: Java-basiert, plattformunabhängig, Nutzung von JDBC für einheitlichen DB-Zugriff, unterstützt Trennung von Layout und Programmlogik
  - JSP-Nachteil: großer Ressourcenbedarf für Java-Laufzeitumgebung
  - PHP-Vorteile: einfache Programmierung durch typfreie Variablen und dynamische Arraystrukturen, fehlertolerant, Automatismen zur Verarbeitung von Formularfeldern, viele Module z. B. für Bezahldienste, XML-Verarbeitung
  - PHP-Nachteile: unterstütze DB-Funktionalität abhängig von jeweiligem DBS; umfangreiche Programmlogik muss als externes Modul (meist in C, C++) realisiert werden

Zusammenfassung

- Cursor-Konzept zur satzweisen Verarbeitung von Datenmengen
  - Operationen: DECLARE CURSOR, OPEN, FETCH, CLOSE
  - Erweiterungen: Scroll-Cursor, Sichtbarkeit von Änderungen
- Statisches (eingebettetes) SQL
  - hohe Effizienz, relativ einfache Programmierung
  - begrenzte Flexibilität (Aufbau aller SQL-Befehle muss zur Übersetzungszeit festliegen, es können nicht zur Laufzeit verschiedene Datenbanken angesprochen werden)
  - SQLJ: eingebettetes SQL für Java
- Call-Level-Interface (z.B. ODBC, JDBC)
  - keine Nutzung eines Präcompilers
  - Einsatz v.a. in Client-Server-Systemen
- JDBC: Standardansatz für DB-Zugriff mit Java
- Stored Procedures: Performance-Gewinn durch reduzierte Häufigkeit von DBS-Aufrufen
  - SQL-Standardisierung: Persistent Storage Modules (PSM)
  - umfassende prozedurale Spracherweiterungen von SQL

Zusammenfassung (2)

- Notwendigkeit dynamisch erzeugter HTML-Seiten mit Zugriff auf Datenbanken
- server-seitige Programmierschnittstellen: CGI, ASP, JSP, PHP ...
- CGI
  - Standardisierte Schnittstelle, die von allen HTTP-Server unterstützt wird
  - pro Interaktion erneutes Starten des CGI-Programms + Aufbau der DB-Verbindung notwendig
  - keine Unterstützung zur Trennung von Layout und Programmlogik
- Java Server Pages (JSP), Active Server Pages (ASP), PHP
  - Einbettung von Programmcode in HTML-Seiten
  - Programmlogik kann in externe Komponenten (z. B. Beans, COM) ausgelagert werden
  - JSP: Verwendung von Java (Servlets) mit Datenbankzugriff über JDBC, SQLJ
  - ASP: auf Microsoft-Server-Plattformen beschränkt; mehrere Programmiersprachen; Datenbankzugriff über ADO (Active Data Objects)
- komplexe Anwendungen erfordern Einsatz von Applikations-Servern