








HPI Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Datenbanksysteme I

Datenbankprogrammierung

19.12.2007

Felix Naumann

SQL mit einer Programmiersprache verbinden

- Embedded SQL
  - Kombiniert SQL mit 7 Programmiersprachen
    - ADA, C, Cobol, Fortran, M, Pascal, PL/I
  - Einbettung von SQL durch Preprocessing
- Stored procedures / PSM
  - Speicherung von Prozeduren als DBMS Objekte
  - Aufruf aus SQL Ausdrücken
- Call-level-interface (CLI)
  - Verbindet C mit DBMS
  - Spezielle Funktionsbibliothek
  - Spart das Preprocessing
- Java Database Connectivity (JDBC)
  - Wie CLI aber für Java
  - Relevant für die Übung

Überblick

- Embedded SQL
- Stored procedures / PSM
- Call-level-interface (CLI)
- Java Database Connectivity (JDBC)

SQL vs. Programmiersprachen

Bisher

- Generische SQL Schnittstelle
- Kommandozeile
- Von allen DBMS angeboten
- Selten genutzt
  - Datenbankadmins

Jetzt

- SQL Ausdrücke in größeren Softwarekomponenten
  - Anwendungen
  - DB Tools
- Verwendet aus einer anderen Programmiersprache heraus

Embedded SQL - Ablauf

HPI Hasso Plattner Institut

Programmiersprache + Embedded SQL

Präprozessor

Programmiersprache + Funktionsaufrufe

Compiler

SQL-Bibliothek (vom DBMS Hersteller)

Ausführbares Programm

Impedance Mismatch

Die Datenmodelle von DBMS und Programmiersprachen unterscheiden sich sehr.

- Programmiersprachen
  - Integer, String, Real, …
  - Pointer
  - Arrays
  - Insbesondere: Keine Mengen
- Relationales DBMS
  - Relationen und Attribute
  - Keine Pointer
  - Keine Schleifen
  - Keine Verzweigungen
  - Keine komplexen Strukturen

Datentransfer zwischen beiden Modellen ist also schwierig.

Impedance Mismatch

- Alternative 1: Nur Programmiersprache verwenden
  - Aber: SQL sehr nützliche und einfache (very high-level) Sprache
  - Aber: Programmier sollen nichts über Speicherstruktur wissen
    - Physische Datenunabhängigkeit!
  - Aber: DBMS sehr effizient
- Alternative 2: Nur SQL verwenden
    - Aber: In Basis-SQL nicht alles ausdrückbar
      - Z.B. n!
    - Aber: Ausgabe sind immer nur Relationen
      - Und z.B. nicht Graphiken
- Alternative 3: Einbettung von SQL in eine Programmiersprache
  - Programmiersprache: „*Host language*“ („Wirtssprache“)
  - SQL: Embedded SQL (eingebettetes SQL)

Die Schnittstelle

- Gemeinsame Variablen (shared variables)
  - Präfix mit Doppelpunkt im SQL Ausdruck
  - Ohne Präfix in Programmiersprache
- Spezielle Umgebung in Programmiersprache
  - **EXEC SQL …**
  - „Warnung“ für den Präprozessor
- Spezielle Variable: SQLSTATE
  - 5 Zeichen lang
  - Für Fehlermeldungen
    - 00000 (keine Fehler)
    - 02000 (Tupel nicht gefunden)
    - …
  - Definiert im SQL Standard

Gemeinsame Variable

- Gemeinsame Variablen deklarieren
  - **`EXEC SQL BEGIN DECLARE SECTION;`**
    `char studioName[50], studioAdr[256];`
    `EXEC SQL END DECLARE SECTION;`
- Syntax entsprechend der Programmiersprache
- Nur primitive Typen verwenden, die beide Sprachen verstehen.
- Verwendung in SQL
  - Wie Konstanten
  - Mit Doppelpunkt als Präfix (**`:studioName`**)
- Verwendung in Programmiersprache
  - Wie jede Variable

Beispiel

- Tupel einfügen
  - ```
    EXEC SQL BEGIN DECLARE SECTION;
            char studioName[50], studioAdr[256];
            char SQLSTATE[6];
    EXEC SQL END DECLARE SECTION;

    /* … Einlesen von Werten in die Variablen … */

    EXEC SQL INSERT INTO Studio(Name, Adresse)
                    VALUES (:studioName, :studioAdr);
    ```
- Klappt für jeden SQL Ausdruck, der kein Ergebnis produziert.
  - Wegen *Impedance Mismatch*
  - **INSERT, DELETE, UPDATE, CREATE, DROP**, …
- Für Anfrage mit Anfrageergebnis
  - Nur ein Tupel: Direkte Bindung an gemeinsame Variablen
  - Mehrere Tupel: Cursor
    ```

Nur ein Tupel: INTO

- *Single-Row-Select*
-
```
void printGehalt() {
EXEC SQL BEGIN DECLARE SECTION;
    char studioName[50]; int vorsitzenderGehalt;
    char vorsitzenderName[50]; char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

/* … Einlesen von studioName … */

EXEC SQL SELECT Gehalt, Manager.Name
                INTO :vorsitzenderGehalt, :vorsitzenderName
    FROM Studio, Manager WHERE VorsitzenderID = ManagerID
    AND Studio.Name = :studioName;
/* Prüfen ob SQLSTATE = 00000 */
/* Gehalt ausgeben */
}
```
- Falls kein Tupel oder mehr Tupel
  - Fehlermeldung
  - Keine Bindung der Variablen
  - Besser: Cursor deklarieren

Cursor

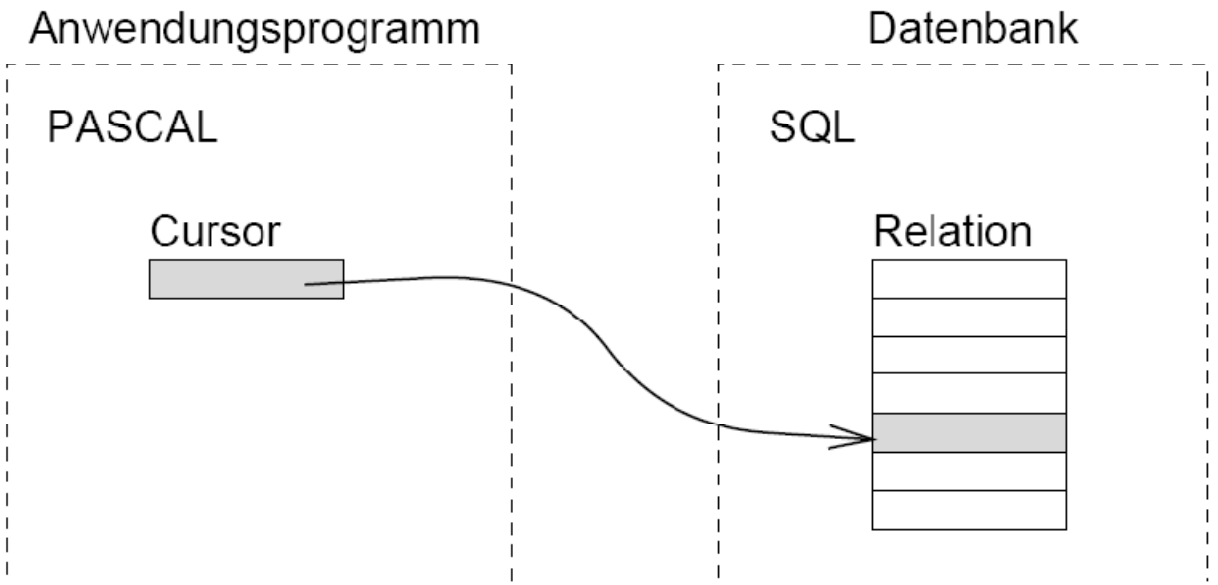


Abbildung: Kai-Uwe Sattler (TU Ilmenau)

Cursor

- Cursor deklarieren
  - **`EXEC SQL DECLARE c CURSOR FOR <Anfrage>;`**
- Öffnen und schließen
  - **`EXEC SQL OPEN CURSOR c;`**
  - **`EXEC SQL CLOSE CURSOR c;`**
- Auslesen
  - **`FETCH`** bewegt Cursor und liest Tupel
  - **`EXEC SQL FETCH c INTO <variablen>;`**
  - Falls kein Tupel gefunden: SQLSTATE = 02000

Cursor – Beispiel


```
void gehaltsBereiche() {
    int i, stellen, counts[15];
    EXEC SQL BEGIN DECLARE SETION;
        int gehalt; char SQLSTATE[6];
    EXEC SQL END DECLARE SETION;
    EXEC SQL DECLARE managerCursor CURSOR FOR
        SELECT Gehalt FROM Manager;

    EXEC SQL OPEN managerCursor;
    for(i=0; i < 15; i++) counts[i] = 0;
    while(1) {
        EXEC SQL FETCH FROM managerCursor INTO :gehalt;
        if(strcmp(SQLSTATE,“02000“)) break;
        stellen = 1;
        while((gehalt /= 10) > 0) stellen++;
        if(stellen <= 14) counts[stellen]++;
        }
    EXEC SQL CLOSE managerCursor;
    for (i=0; i<15; i++)
        printf(„Stellen = %d: Anzahl Manager = %d\n“, i,
counts[i]);
}
```

Scrolling mit Cursor

- Scrolling Cursor deklarieren
  - **EXEC SQL DECLARE c SCROLL CURSOR FOR** <Anfrage>;
- Nach **FETCH** einer dieser Befehle
  - **NEXT** (default)
  - **PRIOR**
  - **FIRST / LAST**
  - **RELATIVE** <integer>
    - **RELATIVE -1** entspricht z.B. **PRIOR**
  - **ABSOLUTE** <integer>
    - **ABSOLUTE 1** entspricht **FIRST**
    - **ABSOLUTE -1** entspricht **LAST**

Dynamisches SQL (*Dynamic SQL*)

Bisher

- Statische SQL Befehle
- Fest vorgegeben
- Werden vom Präprozessor verarbeitet

Jetzt

- SQL Befehle werden zur Laufzeit berechnet
- Präprozessor kennt sie nicht
- Beispiel:
  - Anwendung, die Nutzer nach SQL Anfragen fragt

Dynamisches SQL

- **EXEC SQL PREPARE** SQLAnfrage **FROM** Variable;
  - DBMS kann schon optimieren
- **EXEC SQL EXECUTE** SQLAnfrage;
-
```
void SQLKomandozeile() {
    EXEC SQL BEGIN DECLARE SECTION;
        char *myQuery;
    EXEC SQL END DECLARE SECTION;
    /* … SQL Anfrage einlesen … */
    EXEC SQL PREPARE SQLAnfrage FROM :myQuery;
    EXEC SQL EXECUTE SQLAnfrage;
}
```
- Alternative zu den letzten beiden Zeilen
  - **EXEC SQL EXECUTE IMMEDIATE :myQuery;**
  - Nachteil?

Dynamisches SQL

- Variablen der Programmiersprache in der Anfrage
  - mit ‚**?**‘ in der Anfrage ersetzen
  - Mit **USING** binden
-
```
void filmLoeschen() {
    EXEC SQL BEGIN DECLARE SECTION;
           char Titel [256];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL DECLARE meineAnfrage STATEMENT;
    meineAnfrage = ,DELETE FROM Filme
                              WHERE Titel = ? AND Jahr = ?‘
    EXEC SQL PREPARE SQLAnfrage FROM :meineAnfrage;
    EXEC SQL EXECUTE SQLAnfrage USING :Titel, :Jahr;
}
```

Überblick

- Embedded SQL
- Stored procedures / PSM
- Call-level-interface (CLI)
- Java Database Connectivity (JDBC)

PSM / Stored Procedures

- PSM: Persistent, Stored Modules
  - „Gespeicherte Prozeduren“
  - *Stored Procedures*
- Speicherung von Prozeduren als Datenbankelemente
- Mischung aus SQL und Programmiersprache
- Prozeduren können in SQL Anfragen oder anderen SQL Ausdrücken verwendet werden.
- CREATE PROCEDURE <name> (<parameter>)
  lokale Variablendeklarationen
  body der Prozedur;
- CREATE FUNCTION <name> (<parameter>) RETURNS <Typ>
  lokale Variablendeklarationen
  body der Funktion;

Parameter in PSM

- Drei Teile
  - Modus
    - IN: Nur Input
    - OUT: Nur Output
    - INOUT: beides
    - Bei Funktionen: Nur IN erlaubt
  - Name
  - Typ
-
```
CREATE PROCEDURE Umzug(
    IN alteAdresse VARCHAR(255),
    IN neueAdresse VARCHAR(255)
)
UPDATE Schauspieler
SET Adresse = neueAdresse WHERE Adresse = alteAdresse;
```

CALL zum Aufruf von Stored Procedures

- CALL <prozedurname> (<parameterliste>);
- Beispiele
  - Aus einer Programmiersprache:
    `EXEC SQL CALL Umzug(‚San Diego‘, ‚Los Angeles‘)`
  - Aus einer anderen Stored procedure heraus
  - Als SQL Befehl aus der Kommandozeile
    `Umzug(‚San Diego‘, ‚Los Angeles‘)`
- CALL funktioniert nicht für Funktionen
  - Funktionen nur innerhalb anderer Ausdrücke verwenden

Weitere Befehle

- RETURN <Ausdruck>;
  - Setzt den Rückgabewerte einer Funktion.
  - Beendet nicht Prozedur!
- DECLARE <Name> <Typ>;
  - Lokale Variablen deklarieren
  - Werte bleiben nicht im DBMS erhalten!
- SET <Variable> = <Ausdruck>;
  - Setzt Variablen
  - Ausdruck ‚NULL' ist erlaubt
  - Könnte z.B. eine Anfrage sein (die aber nur einen Wert liefern darf)
- BEGIN … END
  - Gilt als einziger Ausdruck
- Benennung von Ausdrücken
  - Präfix: Name + Doppelpunkt
- Bedingungen deklarieren
  - DECLARE Not_found CONDITION FOR SQLSTATE ‚02000';

IF … THEN … ELSE …


- ```
  IF <Bedingung> THEN
      <Ausdrucksliste>
  ELSEIF <Bedingung> THEN
      <Ausdrucksliste>
  ELSIF
      …
  ELSE
      <Ausdrucksliste>
  END IF;
  ```
- ```
  CREATE FUNCTION SchwarzWeiss(y INT, s CHAR(15)) RETURNS BOOLEAN
  IF NOT EXISTS(
      SELECT * FROM Filme WHERE Jahr = y AND Name = s)
  THEN RETURN TRUE;
  ELSEIF 1 <=
      (SELECT COUNT(*) FROM Filme WHERE Jahr = y AND Name = s AND
  NOT inFarbe)
  THEN RETURN TRUE;
  ELSE RETURN FALSE;
  END IF;
  ```

Anfragen

Viele Verwendungsorte

- Unteranfragen in Bedingungen
  - Siehe vorige Folie
- Falls Anfrage nur einen Wert zurückgibt: Auf der rechten Seite von Wertzuweisungen.
- Als single-row statement wie in Embedded SQL
  - ```
    INSERT INTO Studio(Name, Adresse)
    VALUES (:studioName, :studioAdr);
    ```
  - ```
    CREATE PROCEDURE SomeProc(IN studioName CHAR(15), OUT
    presGehalt INTEGER)
    DECLARE presGehalt INTEGER;
    SELECT Gehalt INTO presGehalt
    FROM Studio, Manager
    WHERE VortsitzenderID = ManagerID AND Studio.name = studioName
    ```
- Mit einem Cursor wie in Embedded SQL
  - Aber ohne EXEC SQL
  - Variablen sind lokal (Doppelpunkt unnötig)
  - Beispiel: Siehe **`PROCEDURE MeanVar`** gleich

Schleifen

- Grundsyntax
  - LOOP <Ausdrucksliste> END LOOP;
- Ausbrechen
  - LEAVE <loopname>
- Weitere Schleifen
  - WHILE <Bedingung> DO <Ausdrucksliste> END WHILE;
  - REPEAT <Ausdrucksliste> UNTIL <Bedingung> END REPEAT;

Beispiel – Cursor und Schleifen

- Gegeben ein Studioname, berechne Mittelwert und Standardabweichung der Filmlängen des Studios
-
```
CREATE PROCEDURE MeanVar(
        IN s CHAR[15],
        OUT mean REAL,
        OUT variance REAL
)
DECLARE Not_Found CONDITION FOR SQLSTATE '02000';
DECLARE FilmCursor CURSOR FOR
        SELECT Laenge FROM Filme WHERE StudioName=s;
DECLARE neueLaenge INTEGER;
DECLARE filmAnzahl INTEGER;
```

Beispiel – Cursor und Schleifen


```
BEGIN
    SET mittelwert = 0.0;
    SET varianz = 0.0;
    SET filmAnzahl=0;
    OPEN FilmCursor;
    FilmLoop: LOOP
           FETCH FilmCursor INTO newLength;
           IF Not_Found THEN LEAVE FilmLoop END IF;
           SET filmAnzahl = filmAnzahl + 1;
           SET mittelwert = mittelwert + neueLaenge ;
           SET varianz = varianz + neueLaenge * neueLaenge;
    END LOOP;
    CLOSE FilmCursor;
    SET mittelwert = mittelwert/filmAnzahl;
    SET varianz = varianz/filmAnzahl - mittelwert *
                                          mittelwert;
END;
```

FOR Schleifen für Cursor

- FOR <loopname> AS <cursorname> CURSOR FOR <Anfrage>
  DO <Ausdrucksliste>
  END FOR;
  - Nur ein Zweck: Cursor
  - Abkürzende Schreibweise (kein Öffnen und Schließen des Cursors, fetching, usw.)
- Beispiel von eben:
- ```
  CREATE PROCEDURE MeanVar(
      IN s CHAR[15],
      OUT mean REAL,
      OUT variance REAL
  )
  DECLARE movieCount INTEGER;
  ```
  ```

FOR Schleifen für Cursor

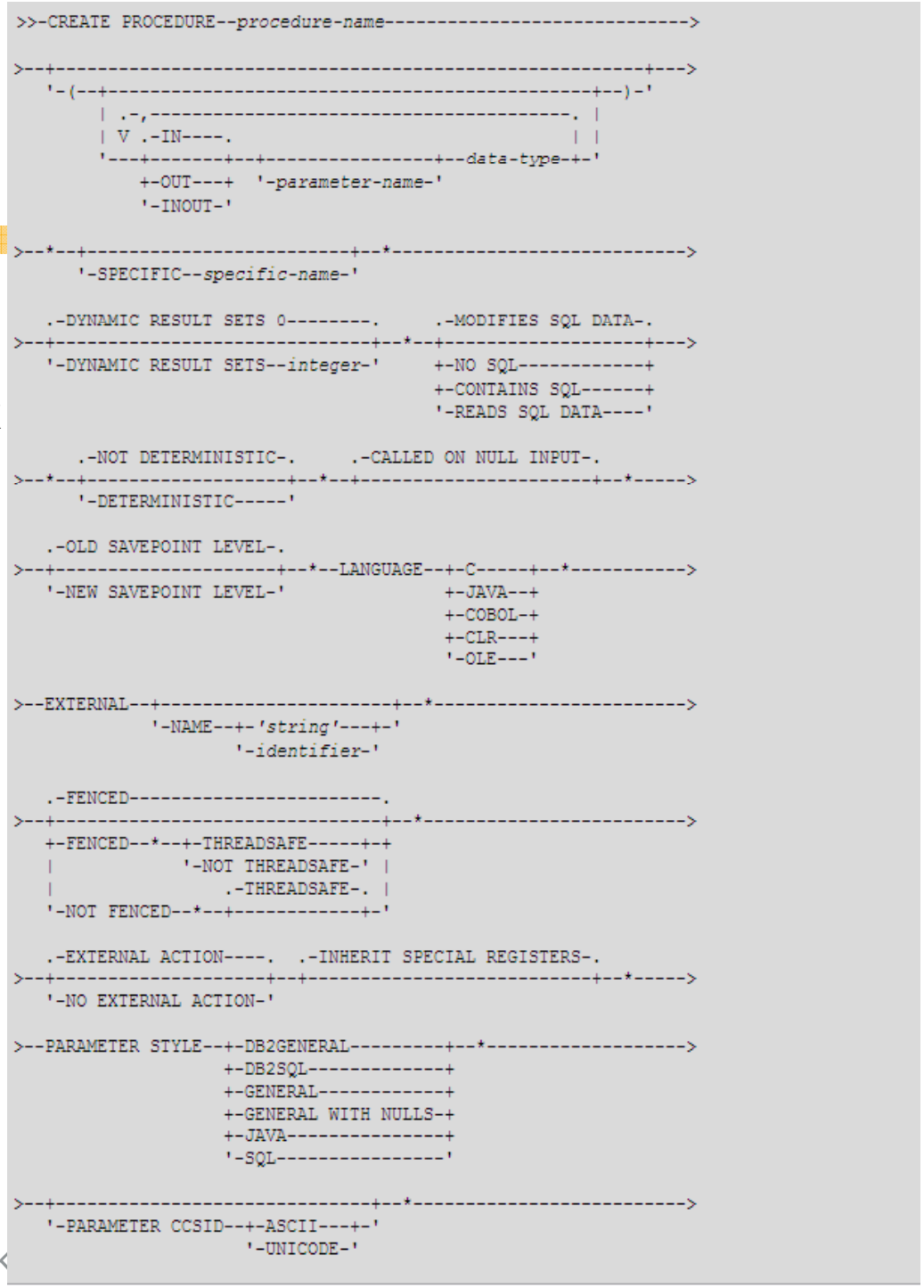

```
BEGIN
    SET mittelwert = 0.0;
    SET varianz = 0.0;
    SET filmAnzahl = 0;
    FOR movieLoop AS FilmCursor CURSOR FOR
            SELECT Laenge FROM Filme WHERE StudioName = s;
    DO
            SET filmAnzahl = filmAnzahl + 1;
            SET mittelwert = mittelwert + Laenge;
            SET varianz = varianz + Laenge * Laenge;
    END FOR;
    SET mittelwert = mittelwert/filmAnzahl;
    SET varianz = varianz/filmAnzahl - mittelwert * mittelwert;
END;
```

Externe Stored Procedures

- Stored Procedures basierend auf einem Stück Code
- Beispiel für DB2

```
CREATE PROCEDURE PARTS_ON_HAND
    (IN PARTNUM INTEGER,
     OUT COST DECIMAL(7,2),
     OUT QUANTITY INTEGER)
EXTERNAL NAME 'parts.onhand'
LANGUAGE JAVA
PARAMETER STYLE JAVA
```

- Code muss in bestimmtem Verzeichnis liegen



Überblick

- Embedded SQL
- Stored procedures / PSM
- Call-level-interface (CLI)
- Java Database Connectivity (JDBC)

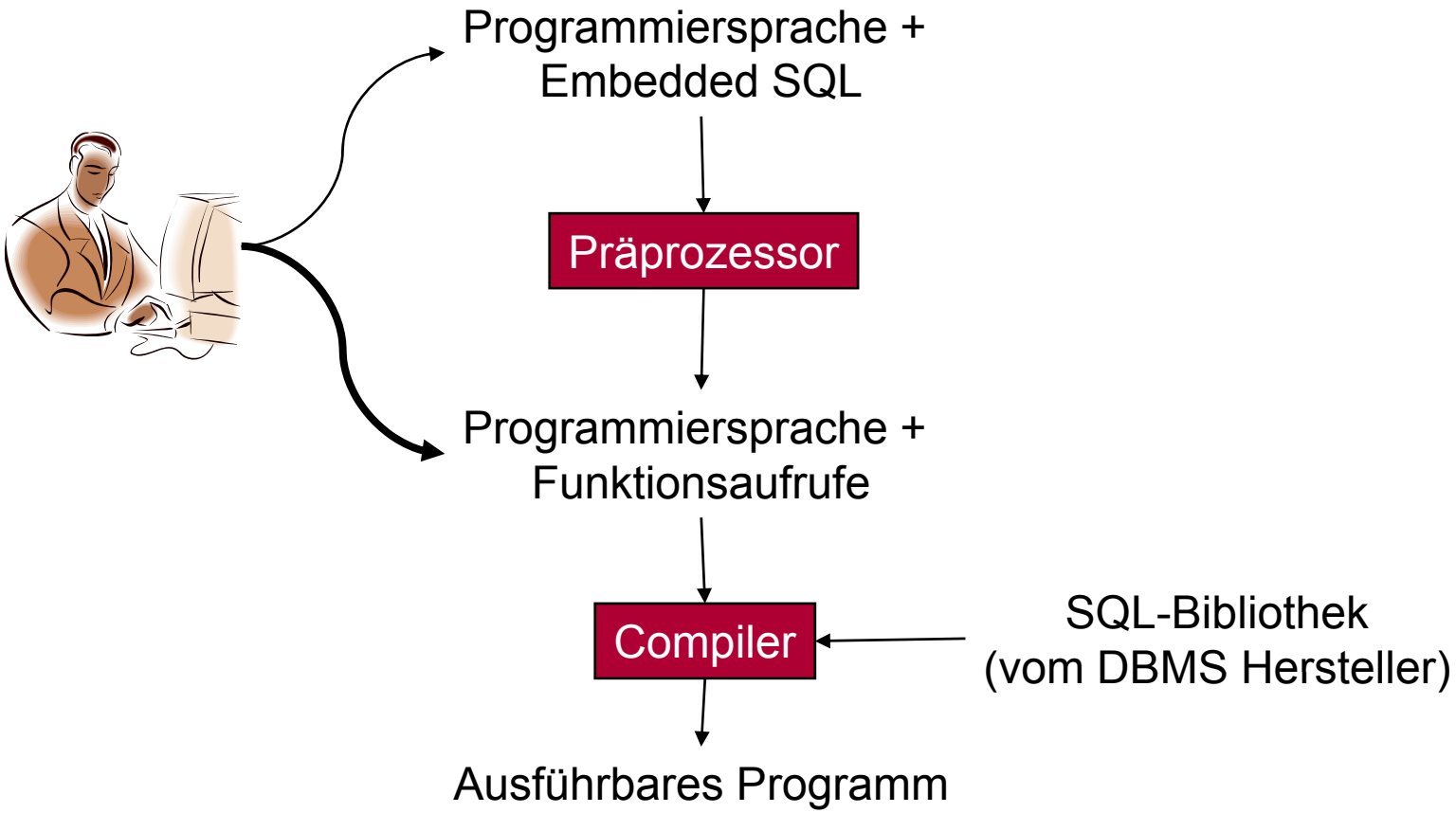
CLI: Call-Level-Interface

Idee

- Programmieren in einer Programmiersprache (Wirtssprache)
- Verwendung spezieller Funktionen für den Datenbankzugriff
  - Funktionsbibliothek
- Umgehung des Präprozessors
  - Kompiliertes Ergebnis ist gleich

- Hier: SQL/CLI
  - Adaptiert von ODBC (Open Database Connectivity)
  - Für die Programmiersprache C
- Später: JDBC (Java Database Connectivity)

Embedded SQL - Ablauf



Datenstrukturen

- Headerdatei sqlcli.h
- Vier Sorten von structs
  - Environments
    - Ein record; von Anwendung erzeugt
    - Repräsentiert das DBMS
  - Connections
    - Ein record zur Verbindung mit einer Datenbank im DBMS
    - Innerhalb einer Environment
  - Statements
    - Anwendung erzeugt ein oder mehr statement records
    - Je eines für jeden SQL Ausdruck
    - Innerhalb einer connection
  - Descriptions
    - Records repräsentieren Tupel (werden vom DBMS gesetzt)
    - oder Parameterlisten (werden von Anwendung gesetzt)

Datenstrukturen

- Jeder record durch einen handle (Pointer) repräsentiert
  - sqlcli.h: **SQLHENV**, **SQLHDBC**, **SQLHSTMT**, **SQLHDES**
- sqlcli.h: **SQL_INTEGER**, **SQL_CHAR**, …
- Erzeugung
  - **SQLAllocHandle(hTyp, hIn, hOut)**
  - **hTyp**: Typ des handles
    - **SQL_HANDLE_ENV**, **SQL_HANDLE_DBC**, **SQL_HANDLE_STMT**
  - **hIn**: handle des jeweils höheren Elements
    - **SQL_NULL_HANDLE** für Environments
  - **hOut**: Adresse des neue Handles

Datenstrukturen - Beispiel


```
#include sqlcli.h
    SQLHENV myEnv;
    SQLHDBC myCon;
    SQLHSTMT execStat;
    SQLRETURN errorCode1, errorCode2, errorCode3;
    errorCode1=SQLAllocHandle(
          SQL_HANDLE_ENV,SQL_NULL_HANDLE,&myEnv);
    if(!errorCode1)
          errorCode2=SQLAllocHandle(
                SQL_HANDLE_DBC,myEnv,&myCo);
    if(!errorCode2)
          errorCode3=SQLAllocHandle(
                SQL_HANDLE_STMT, myCon, &execStat);
```

Anfragen

Idee ähnlich wie bei Dynamic SQL

- **`SQLPrepare(sh, st, sl)`**
  - Statement-Handle **`sh`**
  - SQL Statement **`st`**
  - Stringlänge **`sl`** der Anfrage
    - SQ_NTS falls unbekannt
  - DBMS kann schon optimieren.
- **`SQLExecute(sh)`**
  - Bei INSERT, UPDATE und DELETE alles klar
  - Bei Anfragen: **`SQLFetch(sh)`**
- Beispiele
  - ```
    SQLPrepare(execStat, „SELECT Gehalt FROM Manager“,
     SQL_NTS);
    SQLExecute(execStat);
    ```
  - ```
    SQLExecDirect(execStat, „SELECT Gehalt FROM Manager“,
     SQL_NTS);
    ```

Tupel

- **`SQLFetch(sh)`**
  - Anfrage unter **`sh`** muss bereits ausgeführt sein.
  - Aber: Wo sind nun die Daten?
- **`SQLBindCol(sh, colNo, colType, pVar, varSize, varInfo)`**
  - **`sh`**: Statement handle
  - **`colNo`** und **`colType`**: Spaltennummer und Typ (SQL_CHAR, …)
  - **`pVar`**: pointer auf Variable für den Wert
  - **`varSize`**: Bytelänge der Variable
  - **`varInfo`**: Pointer auf einen Integer für weitere Infos
- **`SQLBindCol`** bindet also eine Spalte an eine Variable. Jedes **`SQLFetch`** verändert die Werte von **`pVar`** und **`varInfo`**.

Beispiel


```
#include sqlcli.h
void worthRanges() {
    SQLHENV myEnv;
    SQLHDBC myCon;
    SQLHSTMT execStat;
    SQLINTEGER gehalt, gehaltInfo;

    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
                           &myEnv);
    SQLAllocHandle(SQL_HANDLE_DBC, myEnv, &myCon);
    SQLAllocHandle(SQL HANDLE STMT, myCon, &execStat);
…
```

Beispiel


```
…
SQLPrepare(execStat, “SELECT Gehalt FROM Manager”,
SQL_NTS);
SQLExecute(execStat);
SQLBindCol(execStat, 1, SQL_INTEGER, &gehalt,
size(gehalt), &gehaltInfo);
while(SQLFetch(execStat)!=SQL_NO_DATA) {
    stellen = 1;
    while((gehalt/=10)>0) stellen++;
    if(stellen<=14) counts[stellen]++;
}
for(i=0;i<15;i++)
    printf(“Stellen=%d:  Anzahl Manager = %d\n”, i,
counts[i]);
}
```

Parameter übergeben

Problem: Parameter aus der Programmierumgebung sollen an Anfrage übergeben werden (wie bei Dynamic SQL).

- Im **SQLPrepare** werden Parameter mit Fragezeichen vorbereitet.
- Mit **SQLBindParameter** werden Werte auf die jeweiligen Stellen gebunden.
  - **SQLBindParameter** hat 10 Parameter…
- Mit **SQLExecute** wird die Anfrage ausgeführt
  - Falls sich Parameter ändern, muss erneut ausgeführt werden.
-

```
/* get values for studioName and studioAddr */
SQLPrepare(myStat,
    “INSERT INTO Studio(name, address) VALUES(?, ?)”, SQL_NTS);
SQLBindParameter(myStat, 1,…, studioName, …);
SQLBindParameter(myStat, 2,…, studioAddr, …);
SQLExecute(myStat);
```

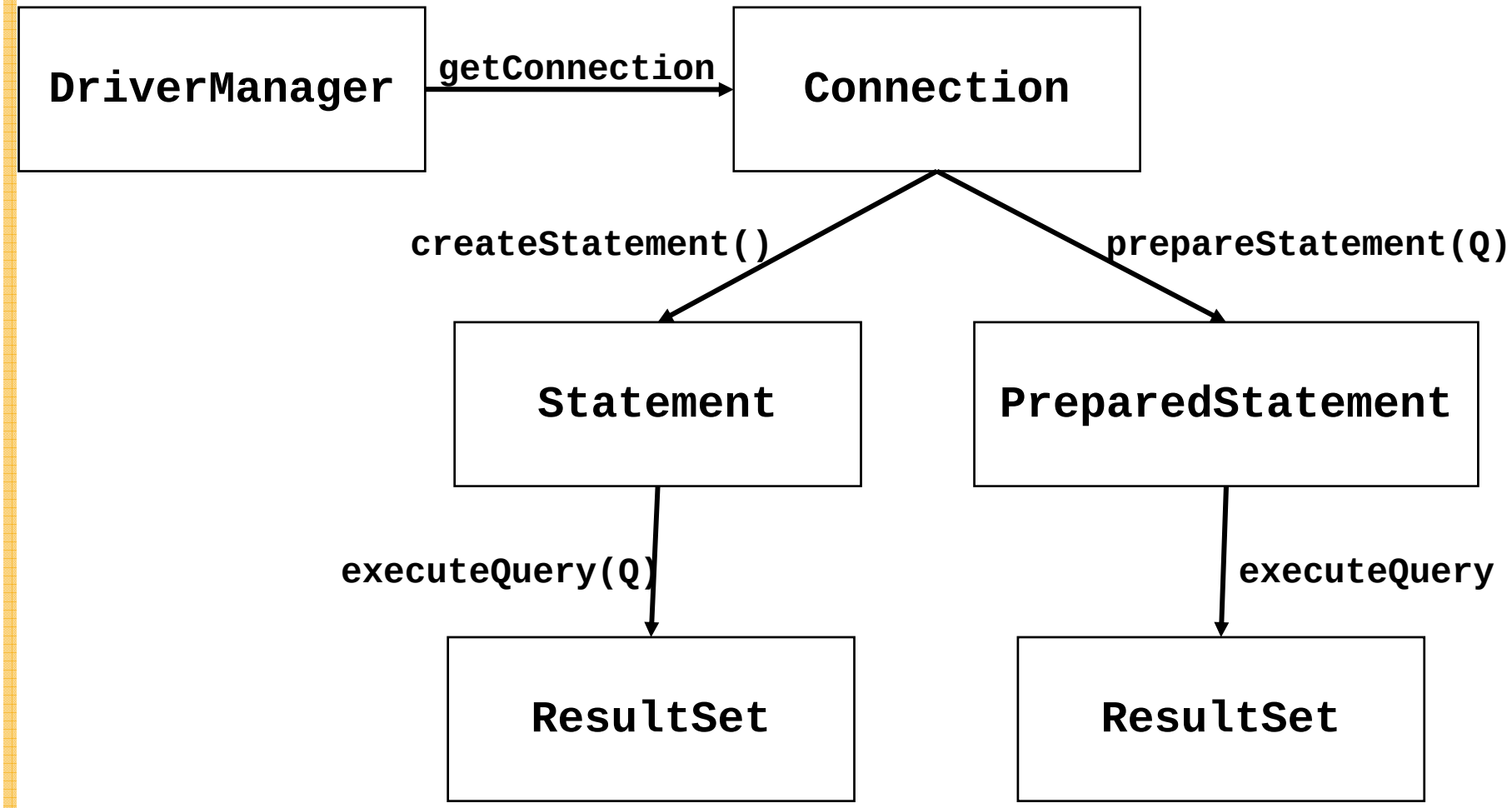
Überblick

- Embedded SQL
- Stored procedures / PSM
- Call-level-interface (CLI)
- Java Database Connectivity (JDBC)

Vergleich zu SQL/CLI

- Gleiches Ziel wie SQL/CLI
- Java statt C als Programmiersprache
- Unterschiede aufgrund der Objektorientierung von Java

Überblick


DriverManager
getConnection
Connection
createStatement()
prepareStatement(Q)
Statement
PreparedStatement
executeQuery(Q)
executeQuery
ResultSet
ResultSet

Erste Schritte

- Treiber für das DBMS einbinden
  - DBMS spezifisch
  - Wird jeweils mit DBMS mitgeliefert
- Verbindung zur Datenbank aufbauen
  - **`Connection meineConnection = DriverManager.getConnection(URL, name, password);`**
  - URL ist DBMS und Datenbank-spezifisch
    - "jdbc:subprotocol:datasource";
  - **`Connection meineConnection = DriverManager.getConnection("jdbc:db2://paprika:50010/FILM1DB", "db2stud","1hugo2");`**

Ausdrücke in JDBC

- Statement
  - **`createStatement();`**
  - Noch keiner SQL-Anfrage assoziiert
- PreparedStatement
  - **`prepareStatement(Anfrage);`**
  - Falls Anfrage öfters ausgeführt wird.
- SQL Ausdrücke ausführen (4 Varianten)
  1. **`ResultSet meinErgebnis = executeQuery(Anfrage);`**
     - ResultSet als Rückgabewert
  2. **`ResultSet meinErgebnis = executeQuery();`**
     - Für PreparedStatements
     - ResultSet als Rückgabewert
  3. **`excuteUpdate(UpdateAnfrage)`**
     - Rückgabewert: Anzahl der veränderten Zeilen als INT
  4. **`excuteUpdate()`**
     - Für PreparedStatements
     - Rückgabewert: Anzahl der veränderten Zeilen als INT

JDBC – Beispiel

- Gegeben die Connection **meineConnection**
- Anfrage: SELECT Gehalt FROM Manager; (2 Varianten der Ausführung)
  1.
```
Statement managerStat = meineConnection.createStatement();
ResultSet gehaelter = managerStat.executeQuery(
                „SELECT Gehalt FROM Manager“);
```
  2.
```
PreparedStatement managerStat =
meineConnection.prepareStatement(
                „SELECT Gehalt FROM Manager“);
ResultSet gehaelter = managerStat.executeQuery();
```
- Updates: Schauspieler einfügen
  -
```
Statement spielerStat = meineConnection.craeteStatement();
spielerStat.executeUpdate(
„INSERT INTO spielt_in VALUES(„ +
„‘Star Wars‘, 1979, ‚Harrison Ford‘ “);
```

Cursor in JDBC (ResultSet)

- Methoden des ResultSet
  - **`next()`**
    - liefert nächstes Tupel
    - FALSE falls kein weiteres Tupel vorhanden
  - **`getString(i)`**
    - Liefert Wert des **`i`**-ten Attributs
    - **`getInt(i)`**, **`getFloat(i)`** usw.
  -
```
while(gehaelter.next()){
        gehalt = gehaelter.getInt(1);
        // gehalt ausgeben o.ä.
}
```

Parameter übergeben

- Mittels **`PreparedStatement`**
- Fragezeichen als Platzhalter für Parameter
  - Bindung mittels **`setString(i, v)`**, **`setInt(i, v)`** usw.
-
```
PreparedStatement studioStat =
meineConnection.prepareStatement(
    „INSERT INTO Studio(Name, Adresse) VALUES(?, ?)“);
// … Werte für studioName und studioAdr vom Nutzer
//   einholen …
studioStat.setString(1, studioName);
studioStat.setString(2, studioAdr);
studioStat.executeUpdate();
```

Zusammenfassung

- Embedded SQL
  - Kombiniert SQL mit 7 Programmiersprachen
    - ADA, C, Cobol, Fortran, M, Pascal, PL/I
  - Einbettung von SQL durch Preprocessing
- Stored procedures / PSM
  - Speicherung von Prozeduren als DBMS Objekte
  - Aufruf aus SQL Ausdrücken
- Call-level-interface (CLI)
  - Verbindet C mit DBMS
  - Spezielle Funktionsbibliothek
  - Spart das Preprocessing
- Java Database Connectivity (JDBC)
  - Wie CLI aber für Java