Kap. 3: Relationale Zugriffsschnittstellen

JDBC

Herangehensweisen

Prozedurale Schnittstelle (Call-Level Interface, CLI):

- Bibliothek, die Funktionen bereitstellt, um mit der DB zu kommunizieren
- Dynamisches SQL: SQL wird als String in der Wirtssprache zusammengebaut
- Wir gucken uns JDBC an; Alternativen: ODBC, OCI, ...

Vorteile:

- dynamisches Binden (String SQL = „SELECT * FROM“ + table -> Sicherheitsprobleme

Nachteil:

- Prüfung erst in dem DBS

Einbettung von SQL in eine Sprache (Syntax-erweiterung einer Sprache)

- SQL wird in die Wirtssprache eingebettet
- Statisches SQL: Zur Übersetzungszeit geprüft (Spezieller Compiler)
- Beispiel Pro*C, SQLJ (ähnlich wie dynamisches... nur ohne deren Methode)

Nachteile:

- SQL ist kein Bestandteil von Programmiersprachen. Daher laeuft es schief.

SQL um prozedurale Elemente erweitern:

- SQL-Programmiersprache
- Beispiel PL/SQL (Oracle)

Nachteil:

- Datenbank-Hersteller abhängig.

Funktionsbibliothek, mit der SQL-Statements zusammengestellt werden

- Beispiel: JOOQ

JDBC

Probleme: Konzeptionelle Unterschiede zwischen Programmiersprache und RDBMS

- Probleme bei der Einbettung von SQL: **Impedance Mismatch**
- Unterschiede: SQL vs. imperative Programmiersprache
  - **SQL**: deklarativ, mengenorientiert *(können groß sein)*
  - **Programmiersprache**: imperativ und Satz-orientiert, komplexes Typ-System mit selbstdefinierten Typen
- Lösungen zur Überwindung des Impedance Mismatch
  - **Iteratoren/Cursor** zur satzweisen Verarbeitung von Ergebnismengen *(Ergebnis Stück für Stück)*
  - **Mapping** der Typen DBMS <-> Programmiersprache *(Numer -> int, long, ..)*
  - **Zugriffsfunktionen** zum Datenaustausch mit Typkonvertierung *(y.B. JDBC)*

JDBC-Treiber

JDBC Thin (Typ 4)

- benutzt Sockets für direkte Verbindung mit Oracle
- beinhaltet eigene Version von Net8 (TCP/IP)
- ist plattform-unabhängig, weil komplett in Java geschrieben

JDBC: Datentypen

Mapping von Java-Typen auf SQL-Datentypen *(Bsp.: String und VARCHAR)*

Handhabung von NULL-Werten

- VARCHAR NULL wird als String null zurückgegeben
- Bei primitiven Datentypen (int etc.) geht das nicht, daher:
- Beim Abfragen:

```
int year = rs.getInt("year");
if (rs.wasNull()) { ... }
```

- Beim Setzen von Werten:

```
stmt.setNull(3, java.sql.Types.NUMERIC);
```

JDBC (Java Database Connectivity)

- Dynamisches SQL: SQL wird als String in Java “zusammengebaut”

JDBC – Aufbau einer Verbindung

- teurere Operation (1x zum Programm start!)

```
private static Connection conn;

public static void main(String[] args) throws SQLException {
    conn = DriverManager.getConnection("jdbc:oracle:thin:@localho", "name", "pw");
    conn.close();
}
```

DDL-Befehl

```
public static void tabelleerstellen() throws SQLException {
    String createOrderItems =
            "CREATE TABLE order_items(" +
                    "     order_id NUMBER(8), " +
                    "     name VARCHAR2(100),"
                    "     PRIMARY KEY (order_id, name))";
    try (Statement stmt = conn.createStatement()) {
        stmt.executeUpdate(createOrderItems);
    }
}
```

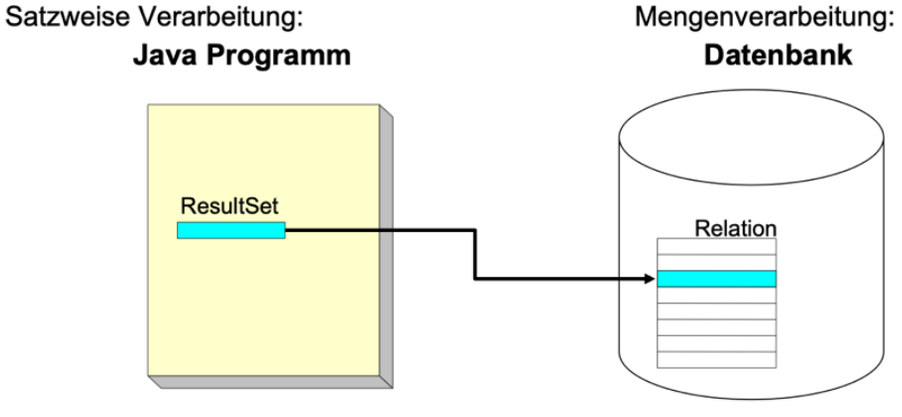
DML-Befehl

- *Wichtig: Rückgabewert von `executeUpdate()` : Anzahl der geänderten Datensätze*

```
public static void prepareStatement (int id, String name) throws SQLException {
    String insertItem = "INSERT INTO rezept VALUES (?, ?)";
    try (PreparedStatement stmt = conn.prepareStatement(insertItem)) {
        stmt.setInt(1, id);
        stmt.setString(2, name);
        stmt.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
public static void createStatement () throws SQLException {
    String insertItem1 = "INSERT INTO order_items VALUES (123,12,'SampleItem1',48.32,12)";
    try (Statement stmt = conn.createStatement()) {
        int num = stmt.executeUpdate(insertItem1);
        System.out.println("Tabelle orderItems "+num+" Zeilen eingefügt!");
    }
}
```

DQL im Detail

```
public static void rezeptAusgeben(int id) throws SQLException {
   try (Statement stmt = conn.createStatement()) {
       String query = "SELECT first_name, last_name, salary "
               + " FROM hr.employees WHERE salary > 5000";
        try (ResultSet rs = stmt.executeQuery(query)) {
            while (rs.next()){
                String last_name = rs.getString("last_name");
                double sal = rs.getDouble(3);
                System.out.println(last_name + "\t" + sal);
            }
        }
    } catch(SQLException e) {
        e.printStackTrace();
    }
}
```



Erzeugung von Primärschlüsseln

Primärschlüssel

- Bedingung: müssen eindeutig sein & unveränderlich

Einfache Realisierungsvariante:

- von (vielen) Datenbanken unterstützte **Sequenzen**
  - zunächst Sequenz in DB erzeugen

```
create sequence employee_seq;
select employee_seq.nextval from dual;
insert into employee values (employee_seq.nextval, ...);
```

JDBC: Probleme

Parameterübergabe / Rückgabewerte

- Positionen der Parameter, Konvertierung der Datentypen

Code ist relativ schlecht zu lesen

- SQL-Code als Strings
- **Vermischung von technischem und fachlichem Code**
- Redundanter Code

```
sql = "SELECT work_hours FROM work_hour WHERE emp_id = ?";
stmt = conn.prepareStatement(sql);
stmt.setLong(1, employee_id);
rs = stmt.executeQuery(); rs.next();
long old_hours = rs.getLong("work_hours");

sql = "SELECT hours_per_day FROM employee " +
      "WHERE emp_id = ?";
stmt = conn.prepareStatement(sql);
stmt.setLong(1, employee_id);
rs = stmt.executeQuery(); rs.next();
long hours_per_day = rs.getLong("hours_per_day");

sql = "UPDATE work_hour SET work_hours = ? " +
      "WHERE emp_id = ?";
stmt = conn.prepareStatement(sql);
stmt.setLong(1, old_hours + hours - hours_per_day);
stmt.setLong(2, employee_id);
stmt.executeUpdate();
```

Beispiel: Regeln für Arbeitszeit

Jeder Mitarbeiter hat eine Soll-Arbeitszeit

Jeder Mitarbeiter hat ein Arbeitszeitkonto

Das Programm errechnet den neuen Stand des Zeitkontos:

- Konto Neu := Konto Alt + geleistete Stunden - Soll-Arbeitszeit

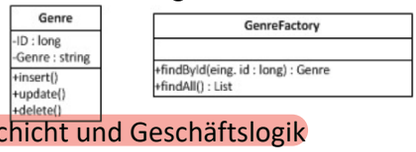
NEUE ALTERNATIVE

- Fachliche Klassen (Entitätsklassen) ermöglichen Zugriff auf die Daten
- Die technischen Details des DB-Zugriffs (SQL, JDBC) sind in der darunterliegenden Persistenzschicht zusammengefasst

```
WorkHours workHours = WorkHoursFactory.findByEmployeeId(employee_id);
Employee employee = EmployeeFactory.findById(employee_id);
workHours.setHours
(workHours.getHours() + hours - employee.getHoursPerDay());
workHours.update();
```

Active Record

- Die Entitätsklasse erhält Methoden, die Datenbankzugriff und SQL- Statements kapseln
- Zusätzlich gibt es eine Factory-Klasse, die neue Objekte aus Datensätzen in der Datenbank erzeugt (oder als statische Methoden in Genre)



Vorteile: Einfaches Muster; einfache Kapselung der SQL-Zugriffe

Nachteile: Immer noch redundanter Code; keine wirkliche Trennung zwischen Persistenzschicht und Geschäftslogik

Performance-Aspekte

- Für alle Überstunden > 30 ein Zusatzgehalt von 50 Euro einstellen

```
List<WorkHour> whs = WorkHourFactory.findAll();
// Alle Überstundendatensätze aller Angestellten durchlaufen
for (WorkHour wh : whs) {
     // Wenn mehr als 30 Überstunden
     if (wh.getHours() > 30) {
                // Pro Überstunde über 30: 50 Euro aufs Gehalt
                ExtraSalary extra = new ExtraSalary();
                extra.setEmployeeId(wh.getEmployeeId());
                extra.setMoney((wh.getHours()-30)*50);
                extra.insert();
       }
}
```

Performanceaspekte: Version 1

```
String sql = "select employee_id, work_hours from work_hour";
PreparedStatement stmt = conn.prepareStatement(sql);
ResultSet rs = stmt.executeQuery();
while (rs.next()) {
       long hours = rs.getLong("work_hours");
       long empId = rs.getLong("employee_id");
       if (hours > 30) {
              sql = "insert into extra_salary values(?, ?)";
              stmt = conn.prepareStatement(sql);
              stmt.setLong(1, empId);
              stmt.setLong(2, (hours-30)*50);
              stmt.executeUpdate();
              stmt.close();
       }
}
```

Performanceaspekte: Version 2

```
String sql = "select employee_id, work_hours from work_hour " +
              "where work_hours > 30";
PreparedStatement sel = conn.prepareStatement(sql);
ResultSet rs = sel.executeQuery();
while (rs.next()) {
       long hours = rs.getLong("work_hours");
       long empId = rs.getLong("employee_id");
       // kein "if (work_hours > 30)" mehr
       sql = "insert into extra_salary values(?, ?)";
       PreparedStatement ins = conn.prepareStatement(sql);
       ins.setLong(1, empId);
      ins.setLong(2, (hours-30)*50);
      ins.executeUpdate();
      ins.close();
}
```

Performanceaspekte: Version 3

```
String sql = "select employee_id, work_hours from work_hour " +
              "where work_hours > 30";
PreparedStatement sel = conn.prepareStatement(sql);
sel.setFetchSize(1000);
ResultSet rs = sel.executeQuery();
sql = "insert into extra_salary values(?, ?)";
PreparedStatement ins = conn.prepareStatement(sql);
while (rs.next()) {
        long hours = rs.getLong("work_hours");
        long empId = rs.getLong("employee_id");

         ins.setLong(1, empId);
         ins.setLong(2, (hours-30)*50);
         ins.addBatch();
}
   ins.executeBatch();
```

Performanceaspekte: Version 4

Komplette Verlagerung in die Datenbank

```
String sql =
    "insert into extra_salary " +
    "select employee_id, (work_hours-30) * 50 " +
    "from work_hour " +
    "where work_hours > 30";
PreparedStatement ins = conn.prepareStatement(sql);
ins.executeUpdate();
```

Performanceaspekte: Fazit

- Durch die Schichtentrennung entsteht ggf. eine ineffiziente Abfolge von Datenbankzugriffen
- In den meisten Fällen ist das ok, d.h. die schlechtere Performance ist in der Praxis nicht schlimm
    - Die Wartbarkeit des Programmes ist viel wichtiger
- Nur bei echten Performance-Problemen sollte hier etwas geändert werden

JDBC: Datenbankunabhängig?

- Jedes DBMS hat eigenen SQL-Dialekt und eigene Typen
- SQL wird von JDBC an die Datenbank weitergegeben. **Folgen**:
  - Verwendung von allen Besonderheiten möglich
  - Führt aber zu Code, der nur mit einem DBMS-Typ arbeitet *(nicht mehr Datenbankunabhängig)*
- Abfrage der Fähigkeiten eines DBMS über JDBC möglich
- Übersetzen von sog. **Escape-Sequenzen** in spezifisches SQL
  - Mapping von Funktionen auf DBMS-spezifische Funktionen
- DBMS-spezifische Typen werden auf Java-Typen gemappt

Sequenzen und Datenbankunabhängigkeit?

- Syntax zur Abfrage einer Sequenz ist (leider) nicht einheitlich; nicht alle Datenbanken unterstützen Sequenzen:

- Oracle:

```
create sequence seq_employee;
select seq_employee.nextval from dual;
insert into employee values (seq_employee.nextval, ...);
```

Postgres:

```
create sequence seq_employee;
select nextval('seq_employee');
insert into employee values (nextval('seq_employee'), ...);
```

Datenbankunabhängigkeit: Funktionen

- DBMS-spezifisches SQL:
  - Oracle:

```
SQL = "insert into employee values (sysdate, ...)"
```

  - Postgres:

```
SQL = "insert into employee values (now(), ...)"
```

  - Lösung:

```
SQL = "insert into employee values ({fn now()}, ...)"
```

  - Dies ist eine JDBC-Escape Sequenz, die vom JDBC-Treiber übersetzt wird.

Zusammenfassung Teil a.)

- DB-Zugriff mit JDBC
- JDBC als API zum Zugriff auf DBMS unterschiedlicher Hersteller
- Treiber; Verbindungsaufbau
- SQL-Befehle werden als Strings an eine API übergeben
- Ausführung von DML und DQL
- Übergabe von Parametern und Rückgabe von Ergebnissen
- Freigabe von Ressourcen

- Erzeugung von **Primärschlüsseln** über Sequenzen
- **Probleme** mit JDBC: Code evtl. schwer zu warten
- Daher: DB-Zugriff sollte in einer Schicht bündeln: Bsp. Active Record
- **Performance**-Aspekte:
  - **Schichtentrennung** führt ggf. zu ineffizienten SQL-Abfolgen.
  - Nur optimieren, wenn nötig
- **Datenbankunabhängigkeit**: Grenzen und Lösungsansätze.

Transaktionen

- **Eine Transaktion ist eine Menge von DB-Operationen, die eine DB von einem konsistenten Zustand in einen weiteren konsistenten Zustand überführt.**
- "Transaktionen sind elementare Ausführungseinheiten, die die Datenbank manipulieren und dabei trotz Mehrbenutzerbetrieb die Korrektheit der ausgeführten Datenbankänderungen und des Datenbankinhalts wahren."

→ Fehlerhafte Datenbankzustände werden ausgeschlossen → Datenbankintegrität

Nutzen von Transaktionen

Mehrbenutzerbetrieb

- Mehrfachzugriff, Nebenläufigkeit
- Aufgabe des DBMS: Inkonsistenzen verhindern

Programmabsturz, DBMS - Absturz, Rechner-Absturz

- Aufgabe des DBMS: Wiederherstellen eines konsistenten, möglichst aktuellen Zustandes

Zwei Aufgabenblöcke für das Transaktionsmanagement

- Synchronisation - Organisation des Mehrbenutzerbetriebs
- Recovery - Wiederherstellung konsistenter Zustände nach Fehlern

Eigenschaften von Transaktionen

Sicht der Anwendung

- Eine Transaktion wird vom Datenbanksystem ganz oder gar nicht durchgeführt ‼️

Sicht des DBMS

- Eine Transaktion überführt die Datenbank von einem konsistenten Zustand in einen weiteren konsistenten Zustand.

Semantische Integrität

- Datenbank-Constraints überwachen

Operationale Integrität

- Konzept der Transaktion als Ausführungseinheit
- Concurrency Control

ACIDèAnforderungen an das DBMS

ACID-Eigenschaft von Transaktionen	
Atomicity	Transaktion ganz oder gar nicht ausführen
Consistency	nach Transaktion ist DB in konsistentem Zustand
Isolation	parallele Transaktionen beeinflussen sich nicht
Durability	Wirkung einer Transaktion ist dauerhaft

:

Parallele Transaktionen

Typisch für Datenbankeinsatz:

- Jedes Programm kann gleichzeitig durch mehrere Benutzer aufgerufen werden, d.h. auf der Datenbank laufen simultan mehrere gleichartige Transaktionen.
- Transaktionen lesen und schreiben Daten aus/in die Datenbank
- Möglicherweise Zugriff auf dieselben Daten
  - Zugriffe müssen synchronisiert werden
- Lese- und Schreiboperationen arbeiten auf Datenbankobjekten (abstrakt!)

Bezeichnung:

- Read(a): Transaktion liest das Objekt a aus der Datenbank
- Write(a): Transaktion schreibt das Objekt a in die Datenbank

Prinzipien bei Transaktionen

- Laufen parallel zueinander ab
- Schritte werden ineinander verschachtelt

Beispielprobleme:

Lost Update

Objekt X hat falschen Wert, da Aktualisierung durch T1 verloren

T1	T2
Read(X);	
X := X-10;	
	Read(X);
	X := X+15;
Write(X);	
Read(Y);	
	Write(X);
Y := Y+10;	
Write(Y);	

Dirty Read

Objekt X muss auf den alten Wert zurückgesetzt werden;
T2 hat aber den falschen Wert schon verarbeitet

T1	T2
Read(X);	
X := X-10;	
Write(X);	
	Read(X);
	X := X+15;
	Write(X);
	Commit;
Read(Y);	
Abort;	

Non-Repeatable Read

Am Ende besitzt X den neuen Wert. Ein Vergleich mit Werten auf Basis des ersten Lesens ist nicht mehr möglich

T1	T2
	Read(X);
	Y := X;
Read(X);	
X := X + 10;	
Write(X);	
Commit;	
	Read(X);
	if (Y != X)
	…

Falsche Summenbildung

T2 liest X nach Subtraktion durch T1 und Y vor Addition; ergibt ein um 10 falsches Ergebnis

T1	T2
	Sum := 0
Read(X);	
X := X-10;	
Write(X);	
	Read(X);
	Sum := sum + X;
	Read(Y);
	Sum := sum + Y;
Read(Y);	
Y := Y+10;	
Write(Y);	

Unterschiedliche Operationenreihenfolge

Beispiel Startwert X=Y=10:
Am Ende besitzt X den Wert 22 und Y den Wert 21, obwohl beide mit den gleichen Operationen bearbeitet wurden

T1	T2
	Read(X);
	X := X+10;
	Write(X);
Read(X);	
X := X*1.1;	
Write(X);	
Read(Y);	
Y := Y*1.1;	
Write(Y);	
	Read(Y);
	Y := Y+10;
	Write(Y);

Phantome

Beispiel:

- T1 liest Kundendatensätze, um alle Kunden herauszufinden, die in Hannover wohnen
- T2 fügt einen neuen Kunden in Hannover-Linden ein
- T1 liest ein zweites Malalle Kundendatensätze, um alle Kunden herauszufinden, die in Hannover - Linden wohnen
- Als Ergebnis von T1 ergibt sich, dass der Neukunde nicht in der ersten, aber in der zweiten Menge enthalten ist, obwohl die zweite eine Teilmenge der ersten sein sollte
- Der eingefügte Neukunde ist für T1 ein Phantom.

JDBC - Transaktionen

Connection-Objekt bestimmt Transaktionssteuerung

Auto-Commit-Modus:

- nach jeder Durchführung eines SQL-Befehls erfolgt automatisch ein commit

Nachteil: mehrere SQL-Befehle lassen sich nicht zu einer Transaktion zusammenfassen

1) für aktive Connection conn wird auto-commit abgeschaltet `conn.setAutoCommit(false);`
2) durchführen aller SQL-Befehle der Transaktion
3) für die Connection explizites commit oder rollback durchführen `conn.commit();` bzw. `conn.rollback();`

Transaktionen

```
try {
   WorkHours workHours =
       WorkHoursFactory.findByEmployeeId(employee_id);
   Employee employee = EmployeeFactory.findById(employee_id);
   workHours.setHours
       (workHours.getHours() + hours - employee.getHoursPerDay());
   workHours.update();
   ConnectionManager.getConnection().commit();
} catch (Exception e) {
   ConnectionManager.getConnection().rollback();
   throw e;
}
```

Beispiel Transaktionssteuerung

```
public static void complexBusinessMethod() throws SQLException {
      boolean ok = false;
      try {
            Person person = new Person();
            ...
            person.insert();

            Movie movie = new Movie();
            ...
            movie.insert();

            ConnectionManager.getConnection().commit();
            ok = true;
      } finally {
            if (!ok)
                ConnectionManager.getConnection().rollback();
      }
}
```

Zusammenfassung Teil b)

- **Mehrbenutzerbetrieb** kann zu inkonsistenten Daten führen:
  - **Synchronisationsprobleme**
- **Abstürze** können zu inkonsistenten Daten in der DB führen
- Lösung: **Transaktionen** als elementare Ausführungseinheiten
- **ACID-Prinzip** beschreibt Garantien des DBMS für die Transaktionen:
  - Atomicity, Consistency, Isolation, Durability

Transaktionssteuerung mit JDBC:

- Auto-Commit abschalten: *Das sollten Sie sich grundsätzlich angewöhnen!*
- Transaktionen starten, wenn ein Befehl an die Datenbank gesendet wird
- Transaktion wird mit `conn.commit()` oder `conn.rollback()` beendet.
- Fehler / Exceptions beachten: Immer für Transaktionsende sorgen!

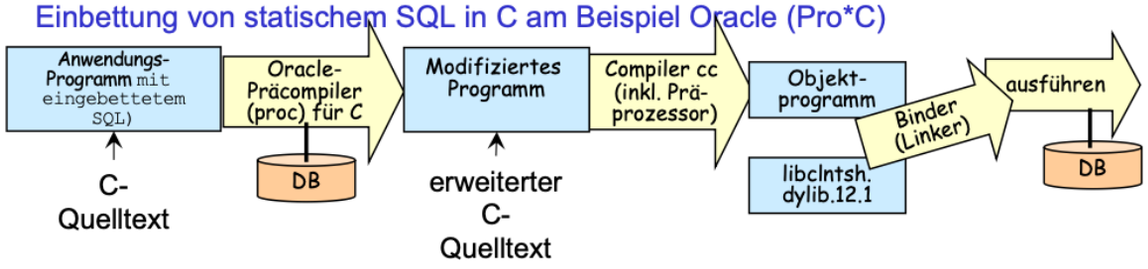
Statisches SQL am Beispiel Pro*C „(Static) Embedded SQL“

- SQL-Statements direkt in Wirtssprache (z.B. C, COBOL, ...) einbetten
- **keine Strings**, die SQL-Befehle enthalten

Erfordert Vorübersetzer (Präcompiler):

- Programme mit eingebetteten DB-Kommandos werden in übersetzungsfähige Programme transferiert.
- Die erforderlichen Präcompiler werden von den Datenbankherstellern geliefert.

Ein Programm enthält also **Anweisungen aus zwei** verschiedenen **Programmiersprachen** (z. B. C und SQL).
Die Einbettung in einige Sprachen (C, Fortran, Cobol) wurde mit SQL92 genormt.



Pro*C: Beispiel

```
#include <stdio.h>
#include <sqlca.h>

char *username = "scott";
char *password = "tiger";
char *database = "orcl";
int hour; int emp_id = 1;

void main()
{
    EXEC SQL CONNECT :username IDENTIFIED BY :password
             USING :database;
    EXEC SQL SELECT work_hours INTO :hour
             FROM work_hour
             WHERE employee_id = :emp_id;
    printf("Aktuelle Ueberstunden: %d\n", hour);
}
```

Embedded SQL in C – Beispiel

- Kein auto-commit, deshalb dann auto-rollback am Ende wenn das vergessen wird

```
void day_finished(long emp_id, long hours)
{
    long new_hours, old_hours, hours_per_day;

    EXEC SQL SELECT work_hours INTO :old_hours
             FROM work_hour
             WHERE employee_id = :emp_id;
    EXEC SQL SELECT hours_per_day INTO :hours_per_day
             FROM employee
             WHERE employee_id = :emp_id;

    new_hours = old_hours + hours - hours_per_day;

    EXEC SQL UPDATE work_hour
             SET work_hours = :new_hours
             WHERE employee_id = :emp_id;
    EXEC SQL COMMIT;
}
```

Probleme bei der Einbettung

Datentypen Programmiersprache / Oracle unterscheiden sich

- Lösung: wie bei JDBC: Mapping definiert

Programmiersprachen kennen (meist) keine NULL-Werte

- Lösung in Pro*C: Indikatorvariablen

Programmiersprachen kennen (meist) keinen Typ "Relation“

- Bzw. Relation zu groß für den Hauptspeicher
- Lösung: Die Verarbeitung von Daten erfolgt satzorientiert.

Hostvariablen - Datentypen

- Pro*C-Datentypen sind die C-Grunddatentypen
- Zusätzlich SQL-spezifische Erweiterung VARCHAR
- Werden vom DBMS auf Oracle-Datentypen abgebildet

Technik: Hostvariablen-Deklaration

- Declare Section nicht notwendig, aber vorteilhaft damit Präcompiler nicht ganzen C-Code analysieren muss.
- Dabei können auch Fehler entstehen, falls der C-Compiler aktueller als der Präcompiler ist.

SQL Statements können sich auf Variablen der Programmiersprache beziehen: „host variables“

- Diese werden in der Anfrage durch einen Doppelpunkt gekennzeichnet.

Deklaration von Hostvariablen:

```
EXEC SQL BEGIN DECLARE SECTION;
  <Datentyp> <Hostvariable>;
  {<Datentyp> <Hostvariable>;}
EXEC SQL END DECLARE SECTION;
```

Behandlung von NULL-Werten

Einsatz von Indikatoren:

- Für jeden Ein-/Ausgabeparameter wird ein zusätzlicher Parameter übergeben.
- Diese Zusatzparameter enthalten Informationen, ob es sich bei dem zugeordneten Parameter um einen NULL-Wert handelt.
- Indikatorvariablen sind Pflicht (sonst Laufzeitfehler bei NULL-Wert)
- Datentyp der Indikatorvariablen in C ist short
- wird mit führendem Doppelpunkt direkt hinter die Hostvariable geschrieben
  (z. B. SELECT ... INTO :plz:plz_i FROM...)
- Indikatorvariable mit Wert -1 zeigt NULL-Wert
  - Achtung: Wert in der Hostvariablen ist dann zufällig und darf nicht verwendet werden!

Das Cursor-Konzept

- Cursor müssen deklariert werden
- Optionen: Sortierung, Veränderbarkeit der Daten

Syntax:

```
<cursor_decl> :=    DECLARE <cursor_name>
                    CURSOR FOR <cursor_spec>

<cursor_spec> :=
                    <table_expr>
                        [ FOR {     READ ONLY |
                                    UPDATE [ OF { <column_name>//, } ] }
                        ]
```

Beispiel:

```
DECLARE c1 CURSOR FOR
SELECT      Name
FROM        Angestellte
WHERE       AbtName = 'Einkauf'
```

Beispiel: Cursor-Zugriff

Deklaration eines Cursors:

```
EXEC SQL
DECLARE cu1 CURSOR FOR
SELECT      Name
FROM        Angestellte
WHERE       AbtNr = :abtnr;
```

Öffnen des Cursors:

```
EXEC SQL OPEN cu1;
```

Auslesen der Daten und Weitersetzen des Cursors:

```
EXEC SQL FETCH cu1 INTO :name;
```

Schließen des Cursors:

```
EXEC SQL CLOSE cu1;
```

Schleife mit Cursor-Zugriff

Wann ist das Ende des Selects erreicht?

```
EXEC SQL OPEN cu1;
for (;;) {
        EXEC SQL FETCH cu1 INTO :name;
}
EXEC SQL CLOSE cu1;
```

Lösung

```
EXEC SQL WHENEVER NOT FOUND <Aktion>

<Aktion> :=
        { CONTINUE | GOTO <label> | STOP | DO <routine> | DO BREAK }
```

Komplettbeispiel

```
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR name[100];
  int hours;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cu1 CURSOR FOR
  SELECT name, hours_per_day FROM employee;

EXEC SQL OPEN cu1;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
for (;;)
{
  EXEC SQL FETCH cu1 INTO :name, :hours;
  name.arr[name.len] = 0;
  printf ("Name: %s Stunden: %d\n", name.arr, hours);
}
printf ("Alle ausgegeben!\n");
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL CLOSE cu1;
```

Dynamisches SQL in Pro*C

- Deklaration einer Variablen für die Anfrage:
  EXEC SQL BEGIN DECLARE SECTION;
  char *anfrage_text;
  int pers_nr;
  EXEC SQL END DECLARE SECTION;
- FestlegungderAnfrage
  anfrage_text = "DELETE FROM employee WHERE employee_id = :empid“;
- Bekannt machen der Anfrage
  EXEC SQL PREPARE anfrage FROM :anfrage_text;
- AusführenderAnfrage
  pers_nr = 1;
  EXEC SQL EXECUTE anfrage USING :pers_nr;

Statisches SQL: Bewertung

Bewertung von statischem SQL

Vorteile

- kompakte Syntax (höhere Benutzerproduktivität), gut lesbar
- Korrektheitsüberprüfung während der Kompilierung
  - korrekte SQL-Syntax
  - korrektes Datenbank-Schema (Semantikprüfung)
  - Typüberprüfungen
  - => höhere Zuverlässigkeit und Robustheit bei Programmausführung

Nachteile

- Relativ altes SQL-orientiertes Konzept
- Z.T. keine Unterstützung in Entwicklungsumgebungen
- Vorübersetzer/Präprozessor benötigt
- Tabellen und Spalten im SQL-Statement sind nicht dynamisch änderbar

Zusammenfassung Teil c)

Statisches SQL (static embedded SQL)

- Grundkonzept: Einbettung von SQL direkt in die Wirtssprache
- Konsequenzen: Compiler-Direktive, Precompiler
- Hostvariablen: Parameterübergabe, Rückgabewerte, Indikatorvariablen
- Cursor-Konzept
- Fehlerbehandlung
- Bewertung: Vor- und Nachteile im Vergleich mit dynamischem SQL

Weitere Ansätze zum Zugriff auf RDBMS

JOOQ: Ideen / Ziele

- Host-Sprache und Abfragesprache besser integrieren
- Typsicherheit, Syntax schon in der IDE prüfen
- Parameterpositionen “sicher”
- SQL-Besonderheiten bestimmter DBMS ausnutzen
    - SQL nicht komplett verbergen (kein O/R-Mapper)
    - Trotzdem Portabilität gewährleisten
- Queries programmatisch zusammenbauen
    - Wir fokussieren uns hier auf diesen Aspekt

JOOQ: Ablauf

Zusammenfassung Teil d)

Beispiel JOOQ

• SQL-Statements programmatisch erzeugen

• Syntax und Schema-Konformität wird vom Compiler geprüft • Dazu ist ein Codegenerator nötig