AG Datenbanken und Informationssysteme

Wintersemester 2006 / 2007

Prof. Dr.-Ing. Dr. h. c. Theo Härder
Fachbereich Informatik
Technische Universität Kaiserslautern

http://wwwdvs.informatik.uni-kl.de

7. Übungsblatt

Für die Übung am Donnerstag, **14. Dezember 2006**,
von 15:30 bis 17:00 Uhr in 13/222.

Aufgabe 1: JDBC und Rekursion

Gegeben sei eine DB mit folgenden Relationen:

TEIL (TNR, TNAME, GEWICHT)

TEILSTRUKTUR (OTNR,UTNR, MENGE)

In der TEIL-Relation befinden sich Teileinformationen wie Teilennummer, -name und -gewicht. Ein Teil kann zum einen aus mehreren Unterteilen bestehen, die jeweils wiederum Unterteile haben können. Zum anderen kann ein Teil ein Unterteil von mehreren Oberteilen sein. Weiterhin sei angenommen, dass ein Teil sich selbst sowohl direkt als auch indirekt nicht enthalten kann. Nur für Teile, die keine Unterteile besitzen, ist ein Gewicht ungleich 0 eingetragen. Die TEILSTRUKTUR-Relation beschreibt die Beziehung zwischen einem Oberteil und Unterteil mit der zugehörigen Mengenanzahl des Unterteils, die in den Oberteil eingeht. Basierend auf den beiden gegebenen Relationen schreiben Sie unter der Anwendung von JDBC ein rekursives Java-Programm, das den für den DB-Zugriff benötigten Benutzernamen und -passwort sowie eine Teilennummer als Eingabeparameter erhält und das Gesamtgewicht des eingegebenen Teils berechnet.

Lösung:

```
    import java.sql.*;

    public class CalcWeight {

      public static float calculateWeight (Connection conn, String otnr)
      throws SQLException {

          Statement stmt1 = conn.createStatement ();
          Statement stmt2 = conn.createStatement ();
          ResultSet rs1, rs2;
          float weight = 0;

          // Hole das Gewicht von otnr
          rs1 = stmt1.executeQuery( "SELECT gewicht FROM teil " +
                                    "WHERE tnr=" + otnr);

          // Hole Unterteile von otnr mit ihrer jeweiligen Mengenangabe
          pstmt2.setString (1, otnr);
          rs2 = stmt2.executeQuery( "SELECT utnr,menge FROM teilstruktur
    " +
                                    "WHERE otnr=" + otnr);

          if (rs2.next ())
          {
              // otnr hat mind. einen Unterteil, d.h. kein eigenes Gewicht
              weight += rs2.getInt(2) *
                  calculateWeight(conn, rs2.getString(1));

              // Berechne dann weitere Unterteile
              while (rs2.next ()) {
                  result += rs2.getInt(2) *
                       calculateWeight(conn, rs2.getString(1));
              }
          }
          else {
              // otnr hat keine Unterteile, hole sein Gewicht
              rs1.next ();
              weight = rs1.getFloat (1);
          }
          rs1.close ();
          rs2.close ();

          stmt1.close ();
          stmt2.close ();

          return (weight);
      }

      public static void main (String []  argv) {

          String url = "..."; // Spezifiziere URL für die JDBC-Verbindung
          String user, passwd;
          String tnr;
          float weight;
          Connection conn;

           // Pruefe Eingabeparameter fuer die DB-Verbindung
           if (argv.length != 3) {
           System.out.println  ("Falsche  Parameter:  <userid>  <passwd>
<part number>.");
```

```
            System.exit (-1);
            }

            user   = argv[0];
            passwd = argv[1];
            tnr    = argv[2];

            try {
            // Lade JDBC-Treiber und  erzeuge Verbindungsobjekt
            Class.forName ("..."); // Spezifiere JDBC-Treiber
            conn = DriverManager.getConnection (url, user, passwd);

            weight = calculateWeight (conn, tnr);
            System.out.println ("Gesamtgewicht von " + tnr + ": " +
weight);

            conn.close ();
            }
            catch (ClassNotFoundException e) {
            System.out.println  ("ClassNotFoundExceptionbeim  Laden  von
JDBC-Driver: " + e.getMessage ());
            }
            catch (SQLException e) {
            System.out.println ("SQLException: " + e.getMessage());
            }
      }
}
```


Aufgabe 2: JDBC und Transaktionen

Eine Flug-DB enthalte eine Relation `SITZPLATZ`, in der Sitzplatzreservierungen von Flügen gespeichert werden und deren Schema durch folgende SQL-Anweisung erzeugt wurde:

```
CREATE TABLE SITZPLATZ(
   FLUGNR       VARCHAR(6),
   DATUM        DATE,
   REIHENNR     SMALLINT,
   PLATZNR      VARCHAR(1),
   KUNDENNAME   VARCHAR(20),
   PRIMARY KEY (FLUGNR, DATUM, REIHENNR, PLATZNR));
```

Es wird angenommen, dass Tupel, die Flugsitzplätze repräsentieren, in der Relation `SITZPLATZ` bereits vorliegen. Hat ein Kunde einen Sitzplatz für einem Flug an einem bestimmten Tag reserviert, so steht sein Name im Attribut `KUNDENNAME` des entsprechenden Tupels. Freie Sitzplätze erkennt man an Tupeln, für die ein Nullwert im Attribut `KUNDENNAME` eingetragen ist.

Ein Reservierungsvorgang für einen Kunden besteht aus den folgenden drei Phasen:

(1) Anzeigen von freien Sitzplätzen bzgl. eines Fluges und Datums

(2) Auswahl eines Sitzplatzes

(3) Belegen des gewählten Sitzplatzes und bestätigen

Schreiben Sie mit Hilfe von JDBC ein Java-Programm, das den oben beschriebenen Reservierungsvorgang im Mehrbenutzerbetrieb realisiert, und setzen Sie dabei das Transaktionskonzept ein. Es soll gewährleistet sein, dass kein Sitzplatz zweimal vergeben wird. Wenn ein Kunde eine Bestätigung für einen Sitzplatz bekommt, dann erhält er diesen garantiert.

Achten Sie auch darauf, dass lange Wartezeiten bei der Durchführung eines Reservierungsvorgangs möglichst vermieden werden. Neben dem Benutzernamen und -passwort soll Ihr Programm die Flugnummer, das Flugdatum sowie den Kundennamen als Eingabeparameter erhalten. Die Sitzplatzwahl soll nach dem Anzeigen der freien Plätze eingelesen werden. Wenn der Platz gebucht ist, erhält der Kunde die Bestätigung.

Lösung:

```
import java.sql.*;

public class Reservation {

  public static void main (String []  argv) {

       String     url = "..."; // URL für die JDBC-Verbindung
       String     user, passwd;
       Connection conn;
       Statement  stmt;
       PreparedStatement pstmt1, pstmt2, pstmt3;
       ResultSet  rs;
       int count;
       Date  fdatum; // Flugdatum
       short rnr;    // Reihennummer
       String fnr, kname, pnr; // Flugnummer, Kundenname und Platznum-
mer

       // Pruefe Eingabeparameter fuer die DB-Verbindung
       if (argv.length != 5) {
```

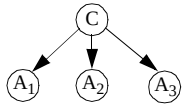
```
            System.out.println ("Falsche Parameterangaben: <userid>
<password> <flight nr> <flight date> <customer name>.");
            System.exit (-1);
        }

        user   = argv[0];
        passwd = argv[1];
        fnr    = argv[2];
        fdatum = Date.valueOf (argv[3]); // Format ist "yyyy-mm-dd"
        kname  = argv[4];

        try {
            // Lade JDBC-Treiber und  erzeuge Verbindungsobjekt
            Class.forName ("..."); // JDBC-Treiber
            conn = DriverManager.getConnection (url, user, passwd);

            // Erzeuge PreparedStatement-Objekte
            pstmt1 = conn.prepareStatement
                ("SELECT reihennr, platznr " +
                 "FROM sitzplatz " +
                 "WHERE flugnr = ? AND datum = ?");

            pstmt2 = conn.prepareStatement
                ("SELECT * " +
                 "FROM sitzplatz " +
                 "WHERE flugnr = ? AND datum = ? AND " +
                 "reihennr = ? AND platznr = ? AND kundenname IS NULL");

            pstmt3 = conn.prepareStatement
                ("UPDATE sitzplatz " +
                 "SET kundenname = ? " +
                 "WHERE flugnr = ? AND datum = ? AND " +
                 "reihennr = ? AND platznr = ?");

            // Phase 1 : Anzeigen von freien Sitzplaetzen
            // Commit wird sofort nach der Ausführung von pstmt1
            // ausgeführt und somit werden Lesesperren auf die gelesenen
            // Daten frei gegeben. Dadurch werden lange Sperr-
            // zeiten vermieden, da während der lange Entscheidungs-
            // phase (nehme ich den Sitzplatz?) keine Sperren ge-
            // halten werden.
            conn.setAutoCommit (true);
            pstmt1.setString (1, fnr);
            pstmt1.setDate (2, fdatum);
            rs = pstmt1.executeQuery ();

            while (rs.next ()) {
                // Ausgabe der freien Sitzplätze
                ...
                }
            // Phase 2: Auswahl eines Sitzplatzes
            // Eingabe der ausgewählten Reihen- und Platznummer
            rnr = ...;
            pnr = ...;

            // Phase 3: Belegung des gewählten Sitzplatzes
            // Überprüfe zuerst , ob zu diesem Zeitpunkt der gewählte
            // Sitzplatz immer noch frei ist. Wenn dies der Fall ist,
dann
            // wird der Sitzplatz belegt. Dieser Lese- und Schreibvor-
gang
            // wird innerhalb einer TA ausgeführt, damit der ausgewählte
```



```
            // Sitzplatz von einem anderen Kunden nicht belegt werden
    kann.
            conn.setAutoCommit (false);
            pstmt2.setString (1, fnr);
            pstmt2.setDate (2, fdatum);
            pstmt2.setShort (3, rnr);
            pstmt2.setString (4, pnr);
            rs = pstmt2.executeQuery ();
            if (! rs.next ()) {
                // Gewählter Sitzplatz ist nicht mehr frei
                // Denkt Kunde während der Auswahlphase zu lange nach,
    so
                // kann der von ihm gewünschten Sitzplatz weg sein, da
    ein
                // anderer Kunde sich schneller dafür entschieden hat.
                // Diesen Nachteil nehmen wir in Kauf, um evtl. eine
    lange
                // Wartezeit, die durch die Denkzeit eines Kunden ver-
    ursacht
                // wird, zu verhindern.
                throw new SQLException ("Sitzplatz ist nicht mehr
    frei.");
            }
            // Update des Tupel zur Sitzplatzreservierung
            pstmt3.setString (1, kname);
            pstmt3.setString (2, fnr);
            pstmt3.setDate (3, fdatum);
            pstmt3.setShort (4, rnr);
            pstmt3.setString (5, pnr);
            count = pstmt3.executeUpdate ();

            conn.commit ();

            // Buchungsbestätigung nach erfolgreichem Commit
            ...

        }

        // Schließe Ressourcen
        pstmt1.close ();
        pstmt2.close ();
        pstmt3.close ();
        conn.close ();
        }
        catch (ClassNotFoundException e) {
            System.out.println ("ClassNotFoundException beim Laden von
    JDBC-Driver: " + e.getMessage ());
        }
        catch (SQLException e) {
            System.out.println ("SQLException: " + e.getMessage());
            // Führe rollback aus
            try {
                    conn.rollback ();
                }
            catch (SQLException ee) {
             System.out.println  ("SQLException  während  Rollback-Ausfüh-
rung: " + ee.getMessage ());
            }
            }
    }
}
```

Aufgabe 3: Verteiltes 2-PC-Protokoll

Betrachten Sie das vollständige Zweiphasen-Commit-Protokoll in der folgenden Umgebung mit einem Koordinator und drei Agenten, die vom Koordinator jeweils in der Reihenfolge A_1, A_2 und A_3 kontaktiert werden:



Welche und insgesamt wieviele Nachrichten werden gesendet und welche Log-Daten werden von den Komponenten geschrieben, wenn der Koordinator den Auftrag für ein Commit erhält und

a) alle drei Agenten ohne Zwischenfall am Commit teilnehmen?

b) der Agent A3 auf das empfangene PREPARE aufgrund eines internen Fehlers mit FAILED antwortet?

c) der Agent A2 auf das empfangene PREPARE ein READY sendet, aber sofort danach abstürzt? Was passiert beim Wiederanlauf des Agenten?

Lösung:

a) Der Koordinator erhält den Auftrag für ein Commit und alle drei Agenten nehmen ohne Zwischenfall am Commit teil.

Koordinator K			Agent A1			Agent A2			Agent A3		
Log	Sende	Empfang	Log	Sende	Empfang	Log	Sende	Empfang	Log	Sende	Empfang
Begin	A1: Prepare A2: Prepare A3: Perpare										
					K: Prepare			K: Prepare			
			Prepared	K: Ready		Prepared	K: Ready				K: Prepare
		A1: Ready A2: Ready							Prepared	K: Ready	
		A3: Ready									
Commit	A1: Commit A2: Commit A3: Commit										
					K: Commit			K: Commit			
			Commit	K: Ack		Commit	K: Ack				K: Commit
		A1: Ack A2: Ack							Commit	K: Ack	
		A3: Ack									
End (async)											

b) Der Koordinator erhält den Auftrag für ein Commit und der Agent A3 antwortet aufgrund eines internen Fehlers mit FAILED.

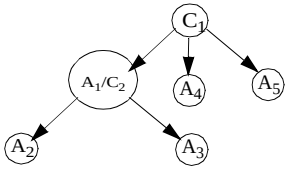
Koordinator K			Agent A1			Agent A2			Agent A3		
Log	Sende	Empfang	Log	Sende	Empfang	Log	Sende	Empfang	Log	Sende	Empfang
Begin	A1: Prepare A2: Prepare A3: Perpare										
					K: Prepare			K: Prepare			
			Prepared	K: Ready		Prepared	K: Ready				K: Prepare
		A1: Ready A2: Ready								K: Failed	
		A3: Failed									
Abort	A1: Abort A2: Abort A3: Abort										
					K: Abort			K: Abort			
			Abort	K: Ack		Abort	K: Ack				K: Abort
		A1: Ack A2: Ack							Abort	K: Ack	
		A3: Ack									
End (async)											

c) Der Koordinator erhält den Auftrag für ein Commit, der Agent A2 antwortet auf das PREPARE mit READY, stürzt aber sofort danach ab.

Koordinator K			Agent A1			Agent A2			Agent A3		
Log	Sende	Empfang	Log	Sende	Empfang	Log	Sende	Empfang	Log	Sende	Empfang
Begin	A1: Prepare A2: Prepare A3: Perpare										
					K: Prepare			K: Prepare			
			Prepared	K: Ready		Prepared	K: Ready				K: Prepare
		A1: Ready A2: Ready				Absturz und Wiederanlauf			Prepared	K: Ready	
		A3: Ready									
Commit	A1: Commit A2: Commit A3: Commit										
					K: Commit						
			Commit	K: Ack							K: Commit
		A1: Ack							Commit	K: Ack	
		A3: Ack									
		A2: Ready					K: Ready				
	A2: Commit										
								K: Commit			
						Commit	K: Ack				
		A2: Ack									
End (async)											

Aufgabe 4: Optimierungen für das 2-Phasen-Commit-Protokoll

In dieser Aufgabe betrachten wir verschiedene Optimierungen für das 2-PC-Protokoll. Gegeben sei folgende Aufrufstruktur zwischen Koordinatoren und Agenten, die in einer hierarchischen Struktur angeordnet sind. Die Agenten werden von C_1 in der Reihenfolge A_1/C_2,A_4, A_5 und von C_2 in der Reihenfolge A_2, A_3 angesprochen.:



Wie viele Nachrichten und Log-Ausgaben werden im Erfolgsfall versendet bzw. geschrieben, wenn:

a) das vollständige hierarchische 2-PC-Protokoll verwendet wird?

b) die spezielle Optimierung für Leser verwendet wird?

c) A_i nach jedem Aufruf in den Prepared-Zustand wechselt?

d) A_i erst beim letzten Aufruf in den Prepared Zustand wechselt?

e) die ACK-Nachricht expizit weggelassen wird?

f) das "spartanische Protokoll" zum Einsatz kommt?

Auf was ist bei den entsprechenden Realisierungsformen zu achten?

Lösung:

a) Nachrichten: (3+2) * 1 (Prepare) + (3+2) * 1 (Ready) + (3+2) * 1 (Commit) + (3+2) * 1 (Ack)
= 4 (Nachrichtenarten) * 5 (Teil-TAs, N)) = 20 Nachrichten

Log-Ausgaben: 2 (Beginn, Committing) * 1 (C1) + 2 (Prepared, Committing) * N (A1,..,A5)
= 2 + 2 * 5 (Teil-TAs, N)
= 2 * 6 = 12 synchrone Log-Nachrichten

Achtung! Es kann zu Blockierungen kommen. z.B. bei Ausfall eines Koordinators.

b) Annahme: 2 Agenten (z.B. A4, A5) werden nur lesend zugegriffen (M=2)

Nachrichten: 4*(3+2) - 2 (Commit, Ack) * 2 (M)
= 20 - 4
= 16 Nachrichten

Log-Ausgaben: (2 + 2 * N) - 2 (Prepared, Committing) * M = 12 - 2 * 2 = 8

c) Nachrichten: 1. Phase fällt flach!
2 (Commit, Ack) * N = 2 * 5 = 10 Nachrichten

Log-Ausgaben: Für jeden Auftrag (Anz Aufträge pro TA = K) in Ai muss
(Prepared, Committing) geloggt werden!!
=> 2 + N * (K+1) = (mit K = 10): 2 + 5 * 11 = 57!

d) Nachrichten: wie bei c) = 10 Nachrichten (gleiche Begründung)

Log-Ausgaben: Prepare mit letztem Auftrag, dann Committing wie immer:
2 + 2 (Prepare, Committing) * N = 2 * (N+1) = 12 synchrone Log-Nachrichten.

Problem: Woher kenne ich den **immer** den letzten Auftrag?
Was passiert, wenn Ai nochmals benötigt wird?

e) Nachrichten: Ack-Nachricht fällt weg.
= 3 (Nachrichtentypen) * 5 (Teil-TAs) = 15

Log-Nachrichten: keine Veränderung zu a)

Vorsicht! Impliziert unendlich langes Gedächtnis des Koordinators

f) Nachrichten: nur noch 1 (Nachrichtentyp, Commit) * 5 = 5 Nachrichten

Log-Nachrichten: wie c) 2 + N * (K+1) = (mit K = 10): 2 + 5 * 11 = 57!
bzw. d) 2 + N * (1+1) = 12

Probleme von d) und e)

Aufgabe 5: Schachtelung von Transaktionen

Können Transaktionen, die bekanntlich ACID-Eigenschaften besitzen, geschachtelt werden?

Begründen Sie Ihre Anwort.

Lösung:

Transaktionen mit ACID-Eigenschaften können nicht geschachtelt werden.

Betrachte folgendes Beispiel:

BOT T_A
...
BOT T_B
Update Tupel t
Commit T_B
...
Rollback T_A

Wird T_A nach dem Commit von T_B zurückgesetzt, so dürfte das Commit von T_B kein "richtiges" Commit sein, denn sonst wäre die Änderung des Tupels t persistent. Handelt es sich jedoch um ein "richtiges" Commit bei T_B, so muss die Änderung von t aufgrund der D-Eigenschaft in der DB dauerhaft bleiben und T_A kann daher nicht zurückgesetzt werden, sondern müsste durch eine weitere Operation kompensiert werden.

Bem.: Siehe Date, 7. Auflage, 471p.