Gruppe A

Bitte tragen Sie **sofort** und **leserlich** Namen, Studienkennzahl und Matrikelnummer ein und legen Sie Ihren Studentenausweis bereit.

PRÜFUNG AUS DATENBANKSYSTEME **VU 184.686**			26. 6. 2014
Kennnr.	Matrikelnr.	Familienname	Vorname

Arbeitszeit: 100 Minuten. Aufgaben sind auf den Angabeblättern zu lösen; Zusatzblätter werden nicht gewertet.

Aufgabe 1: (15)

Kreuzen Sie an, ob die folgenden Aussagen wahr oder falsch sind.

1. Bei der Bearbeitung der Anfrage $\sigma_{A>a}R$ führt ein Hash Index für das Attribut A manchmal zu einer effizienteren Auswertung aber niemals zu einer weniger effizienten Auswertung als ein B+ Baum Index für dieses Attribut.
   wahr ◯ falsch ⊗
2. Betrachten Sie die Kostenformel $2 \cdot b_R \cdot (1 + I)$ mit $I = \lceil \log_{m-1}(\lceil b_R/m \rceil) \rceil$ für das externe Sortieren. Dabei steht I für die Anzahl der "runs".
   wahr ◯ falsch ⊗
3. Die Historie $r_1(C)$, $r_2(D)$, $w_2(D)$, $w_1(D)$, $w_2(C)$, c_1, c_2 vermeidet kaskadierendes Rücksetzen und ist nicht serialisierbar.
   wahr ⊗ falsch ◯
4. Ein geballter Index eignet sich vor allem bei Bereichsabfragen.
   wahr ⊗ falsch ◯
5. Betrachten Sie zwei Relationen $R(ABC)$ und $S(ABD)$. Dann gelten auf jeden Fall folgende Gleichheiten: $(R \ltimes S) = (R \bowtie \pi_A(S)) = (R \bowtie \pi_{AB}(S))$
   wahr ◯ falsch ⊗
6. Eine Relation R sei an 6 Netzwerk-Knoten materialisiert mit den Gewichten 5, 4, 3, 7, 2, und 9. Dann sind $Q_r(R) = 11$ und $Q_w(R) = 19$ gültige Lese- bzw. Schreib-Quoren.
   wahr ◯ falsch ⊗
7. Es gibt Relationen $R(\underline{A}B)$ mit 100 Tupeln und $S(AC)$ mit 100 Tupeln, für die der Ausdruck $R \bowtie S$ 10000 Tupeln ergibt.
   wahr ◯ falsch ⊗
8. Betrachten Sie zwei Relationen $R(AB)$ und $S(AB)$. Dann gelten auf jeden Fall folgende Gleichheiten: $(R \ltimes S) = (S \ltimes R) = (R \bowtie S)$
   wahr ⊗ falsch ◯
9. Mit PL/pgSQL lassen sich alle Anfragen formulieren, die man mit SQL-92 formulieren könnte.
   wahr ⊗ falsch ◯
10. Falls das Rückrollen von Datenbank-Änderungen durch einen Systemabsturz unterbrochen wird, muss das Datenbanksystem beim Wiederanlauf in der Lage sein, das Rückrollen abzuschließen.
    wahr ⊗ falsch ◯

(Pro korrekter Antwort 1.5 Punkte, **pro inkorrekter Antwort -1.5 Punkte**, pro nicht beantworteter Frage 0 Punkte, für die gesamte Aufgabe mindestens 0 Punkte)

Aufgabe 2: Mehrbenutzersynchronisation (14)

In einem DBMS ist eine Datenbank mit den Tabellen A und B implementiert, die als Spalten jeweils eine numerische ID ('id', Primary Key) und einen numerischen Wert ('wert') haben.

Gehen Sie davon aus, dass das DBMS alle Isolation Levels wie folgt implementiert:

Read Uncommited: Striktes 2PL für Exclusive-Locks. Keine Share-Locks.
Read Committed: Striktes 2PL für Exclusive-Locks, Share-Locks werden sofort nach Erhalt wieder freigegeben.
Repeatable Read: Striktes 2PL ohne Einschränkungen.
Serializable: Multiple Granularity Locking ohne Einschränkungen.
Locks sind bis auf Zeilenebene implementiert (row-level locking).

Gegeben sind zwei Transaktionen:

Transaktion 1: SELECT * FROM B; UPDATE A SET wert = 200 WHERE id = 5; COMMIT;

Transaktion 2: UPDATE A SET wert = 300 WHERE id < 10; UPDATE B SET wert = 200; COMMIT;

(a) Bei Isolation Level Serializable kann es hier nie zu einem Deadlock kommen. wahr ◯ falsch ⊗

(b) Bei Isolation Level Repeatable Read kann es zu einem Deadlock kommen, bei Isolation Level Read Committed allerdings nicht. wahr ⊗ falsch ◯

(c) Bei Read Uncommited kann es zu Deadlocks kommen, da die UPDATE-Statements einen Exclusive-Lock auf die selbe Tabelle B anfordern. wahr ◯ falsch ⊗

(d) Wie müssten Sie Transaktion 1 verändern, damit es bei keinem der vier Isolation Levels zu einem Deadlock kommen kann, das Resultat aber das selbe bleibt? (2)

```
UPDATE A SET wert = 200 WHERE id = 5;
SELECT * FROM B;
COMMIT;
```

Zusätzlich zu den Transaktionen 1 und 2 (in der unveränderten Version) wird nun auch folgende Transaktion 3 ausgeführt:

UPDATE B SET wert = 400 WHERE id = 5; UPDATE A SET wert = 300 WHERE id = 5; COMMIT;

(e) Es kann jetzt bei allen Isolation Levels zu einem Deadlock kommen. wahr ⊗ falsch ◯

(f) Es kann bei allen Isolation Levels zu einem Deadlock kommen, wenn nur Transaktion 2 und 3 ausgeführt werden, Transaktion 1 allerdings nicht. wahr ⊗ falsch ◯

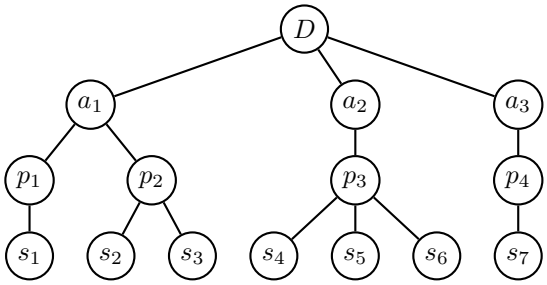
(g) Allein mit Transaktion 2 und 3 können Deadlocks sowohl im Isolation Level Read Committed als auch bei Read Uncommitted auftreten. wahr ⊗ falsch ◯

(h) Bricht Transaktion 2 nach dem zweiten UPDATE-Statement ab und löst einen Rollback aus, so kann es zu kaskadierendem Rücksetzen kommen. wahr ⊗ falsch ◯

(i) Kaskadierendes Rücksetzen kann nur durch zyklische Locks der UPDATE-Statements von Transaktionen 1 und 2 verursacht werden. wahr ◯ falsch ⊗

(Pro korrekter Antwort 1.5 Punkte, **pro inkorrekter Antwort -1.5 Punkte**, pro nicht beantworteter Frage 0 Punkte, für die gesamte Aufgabe mindestens 0 Punkte)

Aufgabe 3: Multi-Granularity Locking (12)
Betrachten Sie folgende Datenbasis-Hierarchie.



Beantworten Sie, welche der folgenden geplanten Sequenzen von Sperr-Anforderungen (bei zwei Transaktionen T_1 und T_2) zu Blockierungen bzw. Deadlocks führen. (Hier bedeutet (T_i, x, L), dass Transaktion T_i versucht, Knoten x in der Hierarchie mit einer Sperre vom Typ L zu belegen.)

Hinweis: Unter Umständen werden nicht alle Sperren dieser Sequenzen auch tatsächlich angefordert, d.h.: Im Falle einer Blockierung einer Transaktion werden die weiter hinten liegenden Sperr-Anforderungen dieser Transaktion gar nicht mehr durchgeführt.

1. (T_1, D, IS), (T_2, D, IX), (T_1, a_1, S), (T_2, a_2, IX), (T_2, p_3, X), (T_1, a_2, S), (T_2, a_1, IX), (T_2, p_1, IX), (T_2, s_1, X):
   Blockierung: ja ⊗ nein ◯ Deadlock: ja ⊗ nein ◯
2. (T_1, D, IS), (T_2, D, IX), (T_1, a_1, IS), (T_1, p_1, IS), (T_2, a_1, IX), (T_2, p_2, X), (T_1, s_1, S):
   Blockierung: ja ◯ nein ⊗ Deadlock: ja ◯ nein ⊗
3. (T_1, D, IX), (T_2, D, IX), (T_1, a_3, IX), (T_2, a_2, IX), (T_1, p_4, IX), (T_2, p_3, IX), (T_1, s_7, X), (T_2, s_5, X):
   Blockierung: ja ◯ nein ⊗ Deadlock: ja ◯ nein ⊗
4. (T_1, D, IX), (T_2, D, IX), (T_1, a_1, IX), (T_2, a_2, X), (T_1, p_2, X), (T_2, a_1, IX), (T_2, p_2, IX), (T_2, s_2, X):
   Blockierung: ja ⊗ nein ◯ Deadlock: ja ◯ nein ⊗

(Pro korrekter Antwort 1,5 Punkte, **pro inkorrekter Antwort -1,5 Punkte**, pro nicht beantworteter Frage 0 Punkte, für die gesamte Aufgabe mindestens 0 Punkte)

Aufgabe 4: Transaktionseigenschaften (4)
Nennen und beschreiben Sie die vier Eigenschaften von *ACID* im Kontext von Datenbankmanagementsystemen.

Atomicity: Transaktion ist kleinste, nicht weiter zerlegbare Einheit; alles oder nichts.

Consistency: Eine Transaktion führt die DB von einem konsistenten Zustand in einen konsistenten Zustand über. D.h. am Ende einer Transaktion müssen alle Konsistenzbedingungen laut Datenbankschema erfüllt sein.

Isolation: Nebenläufige Transaktionen dürfen sich nicht beeinflussen.

Durability: Änderungen erfolgreicher Transaktionen dürfen nicht mehr verloren gehen (auch bei HW/SW-Systemfehlern).

Die folgende Datenbankbeschreibung gilt für die Aufgaben 5 – 8:

Gegeben ist folgendes stark vereinfachtes Datenbankschema, in dem die Spiele der Gruppenphase der Fußball WM abgebildet werden.

`Spiel`(<u>code</u>, gruppe, name)
`Team`(<u>nr</u>, team1: *Spiel.code*, team2: *Spiel.code*)

Auf der letzten Seite dieser Prüfung finden Sie eine Beispielinstanz dieses Schemas!

In der Tabelle `Team` werden die Teams mit ihren eindeutigen Ländercodes (genau drei Buchstaben), der Gruppe (einem Buchstaben) und dem Namen des Landes gespeichert.

In der Tabelle `Spiel` werden die Spiele der Fußball WM gespeichert. Jedes Spiel hat eine eindeutige Nummer `nr`, und zwei Teams, `team1` und `team2`.

Treffen Sie plausible Annahmen bezüglich der Datentypen der Attribute, sofern nicht angegeben.

Aufgabe 5: (6)

Geben Sie CREATE TABLE Statements mit allen nötigen Constraints für die beiden Tabellen an.

```
CREATE TABLE Team (
        code CHAR(3) PRIMARY KEY,
        gruppe CHAR(1),
        name VARCHAR(30)
);

CREATE TABLE Spiel (
        nr INTEGER PRIMARY KEY,
        team1 CHAR(3) REFERENCES Team(code),
        team2 CHAR(3) REFERENCES Team(code)
);
```

Aufgabe 6: (6)

Evaluieren Sie die folgenden SQL-Statements bezüglich der Datenbankinstanz **fußball** (siehe letzte Seite), und geben Sie die Ausgaben der Abfrage an. Falls mehrere Ergebnisse zurückgegeben werden, trennen Sie diese durch ; (Semikolon).

```
SELECT COUNT(*) FROM Team, Spiel;
```

48

```
SELECT COUNT(*) FROM Team t, Spiel s
WHERE t.code=s.team1 OR t.code=s.team2;
```

12

```
SELECT COUNT(*) FROM Team t, Spiel s
WHERE t.code=s.team1 GROUP BY t.code;
```

1 ; 1 ; 2 ; 2

```
SELECT COUNT(*) FROM Spiel s
WHERE s.team1 IN (SELECT code FROM Team);
```

6

Aufgabe 7: (8)

Erstellen Sie einen PL/pgSQL Trigger `checkDuplicates`, der verhindert, dass Spiele hinzugefügt werden können, deren Teams schon einmal aufeinander getroffen sind. In diesem Fall solle die Exception `duplicateMatch` geworfen werden.

Betrachten Sie beispielsweise die Instanz **fußball** auf der letzten Seite:

- Wird versucht, das Spiel `(100, 'ENG', 'ITA')` hinzuzufügen, soll eine Exception geworfen werden, denn das Spiel `(8, 'ENG', 'ITA')` gibt es bereits.
- Wird versucht, das Spiel `(101, 'ITA', 'ENG')` hinzuzufügen, soll ebenfalls eine Exception geworfen werden, denn das Spiel `(8, 'ENG', 'ITA')` gibt es bereits (welches der Teams `team1` und welches `team2` ist, ist egal).
- Wird versucht, das Spiel `(102, 'ITA', 'COL')` hinzuzufügen, soll **keine** Exception geworfen werden, denn es gibt in der Beispielinstanz **fußball** kein Spiel, an dem beide Mannschaften teilgenommen haben.

```
CREATE OR REPLACE FUNCTION fCheckDuplicates() RETURNS TRIGGER AS $$
DECLARE
   duplicateMatch EXCEPTION;
BEGIN
   IF EXISTS (SELECT * FROM Spiel WHERE team1=NEW.team1 AND team2=NEW.team2)
   OR EXISTS (SELECT * FROM Spiel WHERE team1=NEW.team2 AND team2=NEW.team1)
   THEN
      RAISE EXCEPTION duplicateMatch;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER checkDuplicates BEFORE INSERT ON Spiel FOR EACH ROW
EXECUTE PROCEDURE fCheckDuplicates();
```

Aufgabe 8: (10)

Vervollständigen Sie die Java Methode `teamBericht`, der ein Teamcode `code` übergeben wird, und die

- den Namen des Teams ausgibt.
- die Codes aller Teams ausgibt, gegen die dieses Team in einem Spiel spielt.

Wird beispielsweise der Teamcode `ITA` übergeben, so soll (für die Beispielinstanz **fußball**) folgender Text ausgegeben werden:

```
Italien: ENG, CRC, URU
```

Beachten Sie, dass Teamcodes in `team1` oder in `team2` vorkommen können. Erinnern Sie sich daran, dass es in SQL die Möglichkeit gibt, `UNION` zu verwenden. Legen Sie die PreparedStatements hier an. Sie können hier auf eine Connection `c` zugreifen.

```
PreparedStatement psName = c.prepareStatement(
   "SELECT name FROM Team WHERE code=?");
PreparedStatement psCodes = c.prepareStatement(
   "(SELECT team1 FROM Spiel WHERE team2=?) UNION
    (SELECT team2 FROM Spiel WHERE team1=?)");
```

Vervollständigen Sie nun die Methode `teamBericht`. Sie können die oben angelegten PreparedStatements verwenden. Um die genaue Formatierung der Ausgabe und die Fehlerbehandlung brauchen Sie sich nicht zu kümmern.

```
public void teamBericht(String code) throws Exception {
   psName.setString(1, code);
   ResultSet rs1 = psName.executeQuery();
   if (rs1.next()) {
             System.out.println(rs1.getString(1));
   }
   rs1.close();

   psCodes.setString(1, code);
   psCodes.setString(2, code);
   ResultSet rs2 = psCodes.executeQuery();
   while (rs2.next()) {
      System.out.println(rs2.getString(1));
   }
   rs2.close();
}
```

Gesamtpunkte: 75

Sie können diese Seite abtrennen und brauchen sie nicht abzugeben!

Datenbankinstanz **fußball**:

Team		
code	gruppe	name
COL	C	Kolumbien
GRE	C	Griechenland
CIV	C	Elfenbeinküste
JPN	C	Japan
URU	D	Uruguay
CRC	D	Costa Rica
ENG	D	England
ITA	D	Italien

Spiel		
nr	team1	team2
7	URU	CRC
8	ENG	ITA
23	URU	ENG
24	ITA	CRC
39	ITA	URU
40	CRC	ENG