Gruppe A

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

PRÜFUNG AUS DATENBANKSYSTEME **VL 181.186**			13. 1. 2011
Kennnr.	Matrikelnr.	Familienname	Vorname

Arbeitszeit: 120 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. ACID garantiert die Eigenschaften *A*tomicity, *C*onsistency, *I*solation, und *D*urability. wahr ◯ falsch ◯
2. Die Gesamtkosten für die Sortierung einer Tabelle T sind $2b_T(1 + \lceil \log_{m-1}(\lceil m/b_T \rceil) \rceil)$, mit b_T als Anzahl der Seiten von T am Hintergrundspeicher und m als Anzahl der Seitenrahmen im Datenbankpuffer. wahr ◯ falsch ◯
3. PL/(pg)SQL ist eine prozedurale Sprache und spezifiziert daher lediglich was, niemals jedoch wie selektiert werden soll. wahr ◯ falsch ◯
4. Bei der vertikalen Fragmentierung ergeben sich für n Zerlegungsprädikate 2^n Fragmente, die durch einen Join wieder zusammengeführt werden können. wahr ◯ falsch ◯
5. In der relationalen Algebra ist die Operation π distributiv mit $\cup$, $\cap$, und $\setminus$, die Operation σ ist distributiv mit $\cup$. wahr ◯ falsch ◯
6. JDBC ist eine Schnittstelle, die sowohl in Java (per JDBC-Driver) als auch direkt in der Datenbank implementiert sein muss, um zu funktionieren. wahr ◯ falsch ◯
7. Für Relation $R(AB)$ mit 100 Tupeln und Relation $S(\underline{A}C)$ mit 10 Tupeln enthält $\Pi_A(R) - \Pi_A(S)$ mindestens 90 Tupel. wahr ◯ falsch ◯
8. Bei verletzten Deferred Constraints treten Fehler erst am Ende der Transaktion auf. wahr ◯ falsch ◯
9. Wird ein RAID-System zum Speichern der Daten verwendet, so wird dadurch immer auch die Ausfallsicherheit des Systems erhöht. wahr ◯ falsch ◯
10. Wenn ein Join mittels Block Nested Loop Join realisiert wird, kann die Anzahl der benötigten I/O-Operationen im Idealfall weniger sein, als wenn derselbe Join mittels Nested Loop Join realisiert würde. 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]

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: (12)

Gegeben sei ein B^+-Baum vom Typ (k, k^*), wobei k den Verzweigungsgrad in den inneren Knoten und k^* den Grad in den Blattknoten charakterisiert. Jeder Knoten des Baum belegt eine Seite des Hintergrundspeichers, bei gegebener Seitengröße p und Schlüsselgröße s. Verweise (V_i, P, N) innerhalb des Baums haben die Größe v, Blattknoten haben Tupel-IDs (record ids) der Größe d. Für die sequentielle Suche sind die Blattknoten jeweils mit einem Zeiger auf den vorhergehenden (P) und nachfolgenden Blattknoten (N) verbunden.

(a) Bestimmen Sie das maximale k für die inneren Knoten des Baums. [4]

(b) Bestimmen Sie das maximale k^* für die Blattknoten des Baums. [4]

(c) Berechnen Sie das maximale k und k^* wenn $p = 4096$, $s = 4$, $v = 6$, und $d = 8$. [2]

(d) Bestimmen Sie das minimale k und k^* des Baums. [2]

Die folgende Datenbankbeschreibung gilt für die Aufgaben 4 – 8:

Gegeben ist folgendes stark vereinfachtes Datenbankschema, mithilfe dessen die Daten von (Unterhaltungselektronik-) Produkten und Angeboten gespeichert werden.

produkt (<u>pid</u>, name, vorgaenger: *produkt.pid*, topangebot: *angebot.aid*)
angebot (<u>aid</u>, produkt: *produkt.pid*, anbieter, preis)

Jedes Produkt hat eine eindeutige ID *pid*, einen Namen *name*, ein Vorgängerprodukt *vorgaenger* (kann `null` sein) und das beste Angebot für das Produkt *topangebot*.

Jedes Angebot hat eine eindeutige ID *aid*, das Produkt auf es sich bezieht *produkt*, den Händler der es anbietet *anbieter* und einen Preis *preis*.

Wählen Sie entsprechende Datentypen (INTEGER, VARCHAR) für die Attribute.

Aufgabe 4: (8)

Geben Sie die CREATE TABLE Statements mit allen nötigen Constraints für die Tabellen `produkt` und `angebot` an. Beachten Sie eventuelle zyklische Abhängigkeiten zwischen den Fremdschlüsseln. Um die Constraints des Attributs *vorgaenger* brauchen Sie sich nicht zu kümmern.

Geben Sie je ein INSERT Statement für die vorerst leeren Relationen `produkt` und `angebot` an (d.h.: ein Produkt und ein Angebot).

Vergeben Sie plausible Werte für die jeweiligen Attribute, und stellen Sie sicher, dass die Daten in der Datenbank festgeschrieben werden.

Geben Sie die entsprechen DROP Statements für alle zuvor angegebenen CREATE Statements an.

Aufgabe 5: Hierarchische Abfragen (10)

Evaluieren Sie folgendes SQL-Statement mit den angegebenen Tabellen, und geben Sie die Ausgabe der Abfrage an:

produkt			
pid	name	vorgaenger	topangebot
1	'LCD TV 1000'	NULL	101
2	'LCD TV 1050'	1	201
3	'Super LCD TV 1000'	1	302
4	'Super LCD TV 2000'	3	401

angebot			
aid	produkt	anbieter	preis
101	1	'Neptun'	300
102	1	'TV Markt'	400
201	2	'Neptun'	500
301	3	'Neptun'	900
302	3	'TV Markt'	700
401	4	'TV Markt'	400

```
WITH RECURSIVE temp(pid, aid, preis) AS (
    SELECT p.pid, a.aid, a.preis
    FROM produkt p, angebot a
    WHERE p.topangebot = a.aid
    AND vorgaenger IS NULL
UNION ALL
    SELECT p.pid,
        (CASE WHEN a.preis > 2 * t.preis THEN t.aid ELSE a.aid END),
        (CASE WHEN a.preis > 2 * t.preis THEN t.preis ELSE a.preis END)
    FROM produkt p, angebot a, temp t
    WHERE p.topangebot = a.aid
    AND p.vorgaenger = t.pid
)
SELECT * FROM temp;
```

Aufgabe 6: PL/SQL Trigger (8)

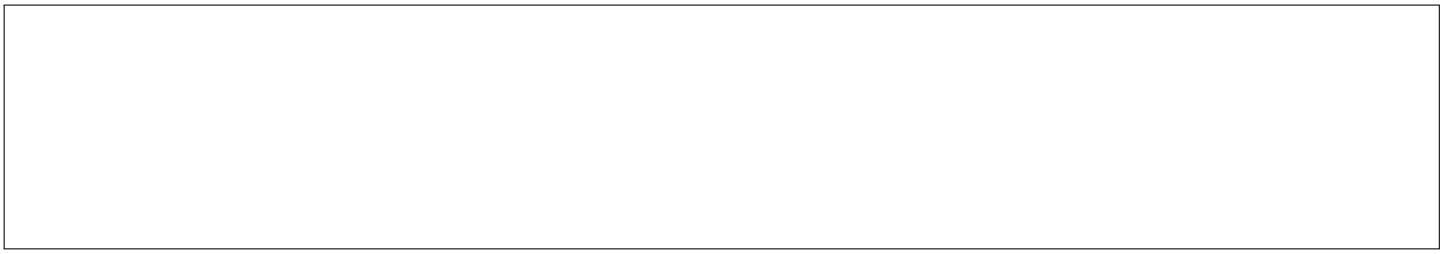
Nehmen Sie an, dass Funktionen und Trigger wie folgt definiert wurden.

```
CREATE FUNCTION fTrigger1()
RETURNS trigger AS $$
BEGIN
    IF (NEW.preis <= 200) THEN
       RETURN NULL;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tTrigger1
BEFORE INSERT ON angebot
FOR EACH ROW EXECUTE PROCEDURE fTrigger1();
```

```
CREATE FUNCTION fTrigger2()
RETURNS trigger AS $$
BEGIN
    IF (OLD.produkt <> NEW.produkt) THEN
        RETURN OLD;
    END IF;
    IF (NEW.preis > OLD.preis * 4) THEN
        NEW.preis = OLD.preis * 2;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tTrigger2
BEFORE UPDATE ON angebot
FOR EACH ROW EXECUTE PROCEDURE fTrigger2();
```

Geben Sie an, wie die Tabelle `angebot` nach jedem der folgenden Befehlsblöcke aussieht.

```
INSERT INTO angebot VALUES (501, 5, 'TV Markt', 1000);
INSERT INTO angebot VALUES (502, 5, 'Neptun', 20);
INSERT INTO angebot VALUES (503, 5, 'Super Tech', 2000);
```



```
UPDATE angebot SET produkt = 4 WHERE aid = 503;
UPDATE angebot SET preis = 5000;
```

Aufgabe 7: (8)

Vervollständigen Sie die folgende Java Methode, die ein Array von Preisen als Input erhält und diese Preise in der Tabelle *angebot* abspeichert.

Verwenden Sie für die INSERT-Befehle unbedingt ein **Prepared Statement**, das in der Schleife jeweils mit den Attribut-Werten des aktuellen Angebots gefüllt wird. Verwenden Sie sowohl als Angebots ID *aid* als auch für das dazugehörige Produkt *produkt* den jeweiligen Array-Index. Nehmen sie als Anbieter *anbieter* den Wert `'Neptun'` an.

Vergessen Sie bitte nicht, Transaktionen abzuschliessen und alle geöffneten Ressourcen auch wieder zu schließen! Verwerden Sie als Verbindungsparameter "jdbc:postgresql://localhost/db", "username","password".

Um eine Fehlerbehandlung und um Java-Imports brauchen Sie sich hierbei nicht zu kümmern. Außerdem wird vereinfachend angenommen, dass die Produkte bereits in der Datenbank vorhanden sind (d.h. Sie brauchen sich nicht um referentielle Integrität zu kümmern).

```
public void neueAngebote(int[] preise) throws Exception {

    Class.forName("org.postgresql.Driver");









}
```

Gesamtpunkte: 75