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**			8. 5. 2013
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. 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 ⊗
2. Bei der vertikalen Fragmentierung ergeben sich für n Zerlegungsprädikate 2^n Fragmente, die durch einen Join wieder zusammengeführt werden können. wahr ◯ falsch ⊗
3. 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 ⊗
4. Wenn ein Join mittels Hybrid Hash 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 ◯
5. Betrachten Sie drei Relationen $U(AB)$, $V(BC)$ und $W(CD)$. Dann gilt auf jeden Fall folgende Gleichheit: $(U \bowtie V) \bowtie W) = (U \times W) \ltimes V)$ wahr ⊗ falsch ◯
6. Bei den Einstellungen steal/¬force ist sowohl ein Undo als auch ein Redo möglich. wahr ⊗ falsch ◯
7. Bei pessimistischen Synchronisationsverfahren können Deadlocks auftreten, die bei optimistischen Verfahren nicht auftreten. wahr ⊗ falsch ◯
8. Wenn ein Unrepeatable-Read-Phänomen auftritt, kann immer auch ein Phantom auftreten. wahr ⊗ falsch ◯
9. ResultSets sind standardmäßig updatable und nur in eine Richtung lesbar (vorwärts). wahr ◯ falsch ⊗
10. Unter Sichtenauflösung versteht man den Vorgang, konstante Ausdrücke und Statements durch möglicherweise vorhandene Views zu ersetzen. 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: Recovery (16)

Gegeben ist die folgende Historie von Transaktionen:

Schritt# i	T_1	T_2	T_3	Log: $[LSN, TA, PageID, Redo, Undo, PrevLSN]$ oder $\langle LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN\rangle$
1	BOT			$[\#1, T1, BOT, 0]$
2		BOT		$[\#2, T2, BOT, 0]$
3			BOT	$[\#3, T3, BOT, 0]$
4	$r(C, c_1)$			
5		$r(B, b_2)$		
6			$r(A, a_3)$	
7	$r(A, a_1)$			
8		$w(C, b_2 * 5)$		$[\#4, T2, P_C, C+ = 700, C- = 700, \#2]$
9	$w(B, a_1 + c_1)$			$[\#5, T1, P_B, B+ = 200, B- = 200, \#1]$
10			$w(A, a_3 + 250)$	$[\#6, T3, P_A, A+ = 250, A- = 250, \#3]$
11	$r(B, b_1)$			
12	$w(C, b_1 - a_1)$			$[\#7, T1, P_C, C- = 700, C+ = 700, \#5]$
13		commit		$[\#8, T_2, commit, \#4]$ .....
14	commit			$[\#9, T_1, commit, \#7]$ .....
15			`rollback` .	$[\#10, T_3, rollback, \#6]$ .....
16				$\langle \#11, T_3, P_A, A- = 250, \#10, \#3\rangle$ ....
17				$\langle \#12, T_3, (BOT), \#11, 0\rangle$ .....

(a) Nehmen Sie an, dass in Zeile 15 auch die Transaktion T_3 mittels *commit* beendet wird. Ist die resultierende Historie serialisierbar?

◯ Ja ⊗ Nein

Wenn ja, in Reihenfolge T- ..... vor T- ..... vor T- .....

Wenn nein: die Historie wird durch das Streichen von zumindest `1` ..... Operationen serialisierbar.

(b) Führen Sie nun – unabhängig davon, ob die Historie serialisierbar ist – in Zeile 15 ein rollback für T_3 durch.

Zu Beginn ist der relevante Datenbestand in der Datenbank A = 100, B = 200 und C = 300. Tragen Sie nun das Recovery-Log zu dieser Historie (mit *rollback* von T_3 in Zeile 15) in die rechte Spalte der obigen Tabelle ein. Dabei sind Undo/Redo-Einträge *relativ* zum Datenbestand anzugeben. Geben Sie in den Zeilen 15 ff. die Log-Einträge für die Recovery an.

Die Werte von A, B und C nach dem rollback von T_3 sind $A:$ `100` ....., $B:$ `400` ....., $C:$ `300` .....

Die folgende Datenbankbeschreibung gilt für die Aufgaben 4 – 7:

Gegeben ist folgendes stark vereinfachtes Datenbankschema, in dem die Kartendaten eines Kreditkartenunternehmens gespeichert werden.

`Kartentyp`(tid, limit)
`Karte`(kid, tid: *Kartentyp.tid*)
`Umsatz`(uid, kid: *Karte.kid*, betrag, jahr, monat)

Im `Kartentyp` werden die verschiedenen Arten der Kreditkarten zusammen mit einem Limit gespeichert. Das Limit darf vom Kunden pro Karte und Monat nicht überschritten werden (die Nichtüberschreitung des Limits muss aber *nicht* per Constraint sichergestellt werden!)

Die Tabelle `Karte` enthält die Zuordnung von Kartennummer und Kartentyp.

In der Tabelle `Umsatz` werden alle getätigten Zahlungen protokolliert.

Treffen Sie plausible Annahmen bezüglich der Datentypen der Attribute.

Aufgabe 4: (6)

Geben Sie CREATE TABLE Statements mit allen nötigen Constraints für die drei Tabellen an. Treffen Sie plausible Annahmen für die Datentypen der Attribute.

Geben Sie je ein INSERT Statement für jede der drei Tabellen an. Vergeben Sie plausible Werte für die jeweiligen Attribute.

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

```
CREATE TABLE Kartentyp (
  tid NUMBER(10) PRIMARY KEY,
  limit NUMBER(20)
);

CREATE TABLE Karte (
  kid NUMBER(10) PRIMARY KEY,
  tid NUMBER(10) REFERENCES Kartentyp(tid)
);

CREATE TABLE Umsatz (
  uid NUMBER(10),
  kid NUMBER(10) REFERENCES Karte(kid),
  betrag NUMBER(10),
  jahr NUMBER(4),
  monat NUMBER(2),
  PRIMARY KEY (uid, kid)
);

INSERT INTO Kartentyp VALUES (1,1000);
INSERT INTO Karte VALUES (1,1);
INSERT INTO Umsatz VALUES (1,1,1000,2013,1);

DROP TABLE Umsatz;
DROP TABLE Karte;
DROP TABLE Kartentyp;
```

Aufgabe 5: (6)

Evaluieren Sie folgende SQL-Statements bezüglich der angegebenen Tabelle, und geben Sie die Ausgaben der Abfragen an:

Umsatz				
uid	kid	betrag	jahr	monat
1	1	400	2013	01
2	1	100	2013	01
3	1	400	2013	02
4	1	300	2013	03
1	2	200	2013	01
2	2	400	2013	01
3	2	500	2013	02
4	2	100	2013	02
5	2	200	2013	03
1	3	300	2013	02
2	3	600	2013	02
3	3	700	2013	03

```
SELECT u1.jahr, u1.monat, u1.kid, SUM(u1.betrag)
FROM Umsatz u1
GROUP BY u1.jahr, u1.monat, u1.kid
HAVING SUM(u1.betrag) >= ALL (
    SELECT SUM(u2.betrag)
    FROM Umsatz u2
    WHERE u2.jahr = u1.jahr and u2.monat = u1.monat
    GROUP BY u2.jahr, u2.monat, u2.kid
)
```

Ergebnis d. Abfrage

jahr	monat	kid	summe
2013	1	1	500
2013	2	3	900
2013	3	3	700

```
SELECT COUNT(*) FROM Umsatz u1, Umsatz u2;
```

144

Aufgabe 6: (9)

Erstellen Sie einen PL/pgSQL Trigger `checkLimit`, der vor dem Einfügen von neuen Umsatz-Datensätzen prüft, ob mit diesem Umsatz das Monatslimit der Karte überschritten wird (d.h. bisheriger Monatsumsatz der Karte und neuer Umsatz zusammen überschreiten das Limit). Falls dem so ist, soll eine Exception geworfen werden. Die Exception soll vom Aufrufer behandelt werden, und darf daher nicht im Trigger behandelt werden.

```
CREATE FUNCTION fCheckLimit()
RETURNS trigger AS $$
DECLARE
  limitExceeded EXCEPTION;
  actSum NUMBER;
  limit NUMBER;
BEGIN
  SELECT SUM(betrag) INTO actSum
      FROM Umsatz WHERE kid = NEW.kid and jahr = NEW.jahr and monat = NEW.monat;
  SELECT t.limit INTO limit
      FROM Karte k,Kartentyp t WHERE k.kid = NEW.kid and k.tid = t.tid;
  IF ((NEW.betrag + actSum) > limit) THEN
    RAISE limitExceeded;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER checkLimit
BEFORE INSERT ON Umsatz
FOR EACH ROW EXECUTE PROCEDURE fCheckLimit();
```

Aufgabe 7: (9)

Vervollständigen Sie die folgende Java Methode `calcFee`, welche die Kartengebühr berechnet. Die Kartengebühr in Prozent ergibt sich aus dem Umsatz pro Monat: $100 - (\frac{Umsatz\ des\ Monats}{Kartenlimit} \cdot 100)$. D.h. je mehr verbraucht wird desto billiger ist die Karte. Wird das Kartenlimit vollständig ausgeschöpft, fällt keine Kartengebühr an. Wird eine Karte in einem Monat nicht benützt, fallen 100% Kartengebühr an.

Die Methode `calcFee` bekommt eine `Connection`, das Jahr, den Monat und die Karten-ID übergeben, und soll den berechneten Prozentsatz zurückgeben (100% soll dem Wert `1` entsprechen, 33% entspricht beispielsweise dem Wert `0.33`).

Stellen Sie sicher, dass die Methode alle darin geöffneten Ressourcen wieder frei gibt (aber nicht die übergebene `Connection`). Sie brauchen sich nicht um Fehlerbehandlung zu kümmern.

Hinweis: Um die Arbeit so einfach wie möglich zu gestalten, empfiehlt es sich, die Berechnung des Rückgabewerts weitestgehend auf Datenbankebene durchzuführen!

```
public float calcFee(Connection c, int year, int month, int card) throws Exception {
    float fee = 100;
    Statement stmt = c.createStatement();
    CallableStatement cstmt = stmt.prepareStmt("SELECT 100-((SUM(u.betrag)/t.limit)*100) " +
        "FROM Kartentyp t, Karte k, Umsatz u " +
        "WHERE u.kid = ? and u.jahr = ? and u.monat = ? and u.kid = k.kid and k.tid = t.tid " +
        "GROUP BY t.limit");
    cstmt.setInt(1,card);
    cstmt.setInt(2,year);
    cstmt.setInt(3,month);
    ResultSet rs = cstmt.executeQuery();
    if(rs.next()) {
        fee = rs.getFloat(1);
    }
    rs.close();
    cstmt.close();
    stmt.close();
    return fee;
}
```

Gesamtpunkte: 75