Gruppe A	PRÜFUNG AUS DATENBANKSYSTEME **VL 181.186**		22. 4. 2010
Kennnr.	Matrikelnr.	Familienname	Vorname

Arbeitszeit: 120 Minuten. Aufgaben sind auf den Angabeblättern zu lösen; Zusatzblätter werden nicht gewertet.

Aufgabe 1: (15)

Gegeben ist die folgende Historie von Transaktionen:

Schritt	T_1	T_2	T_3	Log: [LSN, TA, PageID, Redo, Undo, PrevLSN] or ⟨LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN⟩
1	BOT			[#1, T_1, BOT, 0]
2		BOT		[#2, T_2, BOT, 0]
3			BOT	[#3, T_3, BOT, 0]
4	$r(C, c_1)$			
5		$r(C, c_2)$		
6			$r(A, a_3)$	
7		$w(A, c_2 * 2)$		[#4, T_2, P_A, A+=100, A-=100, #2] ....
8	$w(A, c_1 + 200)$			[#5, T_1, P_A, A+=100, A-=100, #1] ....
9			$r(B, b_3)$	
10			$w(B, a_3 + b_3)$	[#6, T_3, P_B, B+=100, B-=100, #3] ....
11		$w(C, c_2 + 200)$		[#7, T_2, P_C, C+=200, C-=200, #4] ....
12		$r(B, b_2)$		
13		$w(B, b_2 + c_2)$		[#8, T_2, P_B, B+=100, B-=100, #7] ....
14	commit			[#9, T_1, commit, #5] .....
15		commit		[#10, T_2, commit, #8] .....
16			rollback ...	[#11, T_3, rollback, #6] .....
17				⟨#12, T_3, P_B, B-=100, #11, #3⟩ .....
18				⟨#13, T_3, (BOT), #12, 0⟩ .....

(a) Nehmen Sie an, dass in Zeile 16 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 16 ein *rollback* für T_3 durch.

Zu Beginn ist der relevante Datenbestand in der Datenbank $A = 100$, $B = 200$ und $C = 100$. Tragen Sie nun das Recovery-Log zu dieser Historie (mit rollback von T_3 in Zeile 16) in die rechte Spalte der obigen Tabelle ein. Dabei sind Undo/Redo-Einträge *relativ* zum Datenbestand anzugeben. Geben Sie in den Zeilen 16 ff. die Log-Einträge für die Recovery an.

Die Werte von A, B und C nach dem rollback von T_3 sind [A : 300 ..... ; B : 300 ..... ; C : 300 .....].

Aufgabe 2: (15)

Nehmen Sie an, dass ein DBMS Tupel aus einer Datei (Heap File) von der Festplatte nachladen und sortieren muss. Das DBMS verwendet dazu externes Sortieren (External Multi-Way Merge Sort) unter effizienter Nutzung des verfügbaren Datenbankpuffers.

In der Datei befinden sich 4500 Tupel. Der Sortierschlüssel für die Datei ist 4 Bytes lang. Tupel IDs sind 8 Bytes und Seiten IDs sind 5 Bytes lang. Jedes Tupel ist in Summe 48 Bytes lang. Eine Seite ist 512 Bytes groß, wobei jede Seite 12 Bytes an Kontrollinformation verbraucht. Der Datenbankpuffer besteht aus 4 Seiten.

(a) Wenn das DBMS den Multi-Way Merge Sort Algorithmus zum externen Sortieren anwendet, wie viele sortierte Runs gibt es nach dem initialen Pass, und wie groß ist jeder dieser Runs? Sind alle Runs gleich groß? Gehen Sie davon aus, dass ein Tupel nicht auf mehrere Seiten aufgespalten werden kann. [3]

Pro Seite sind 500 (512 - 12) Bytes zum Speichern von Tupeln vorhanden. Jede Seite kann maximal 10 Tupel zu je 48 Bytes speichern (Aufspalten ist laut Angabe nicht möglich). Somit werden 450 Seiten gebraucht, um die 4500 Tupel zu speichern. Bei 4 Pufferseiten gibt es somit $\lceil 450/4 \rceil = 113$ sortierte Runs, 112 je 4 Seiten lang, der letzte Run hat nur 2 Seiten $(450 - 112 \cdot 4)$.

(b) Wieviele Passes (inklusive dem zuvor betrachteten initialen Pass) braucht es, die Datei komplett extern zu sortieren? [2]

Man braucht insgesamt $\lceil \log_3 113 \rceil + 1 = 6$ Passes.

(c) Was sind die gesamten I/O Kosten zum externen Sortieren dieser Datei? [2]

Gesamtkosten: $2 \cdot 450 \cdot 6 = 5400$ I/Os.

(d) Was ist die größte (wenn man die Anzahl von Tupel betrachtet) Datei, die man mit 4 Pufferseiten in 2 Passes sortieren kann? Was, wenn man 257 Pufferseiten zur Verfügung hat? [3]

Im Initial Pass werden $\lceil N/m \rceil$ Runs erstellt, wobei N die Anzahl der Seiten und m die Anzahl der Pufferseiten bezeichnet. Im ersten nachfolgenden Pass muss man diese Runs mergen können, d.h., $m - 1 \geq \lceil N/m \rceil$. Bei $m = 4$ ergibt das $N = 12$ Seiten, und somit $12 \cdot 10 = 120$ Tupel. Bei $m = 257$ bekommt man $N = 256 \cdot 257 = 65792$ mit 657920 Tupel.

(e) Nehmen Sie an, dass Sie einen B+ Baum Index haben, dessen Suchschlüssel dem Sortierschlüssel aus dem Heap File gleicht. Der Auslastungsgrad des Baumes (Nutzdaten in den Blättern) sei 60 Prozent, und die internen Navigationsknoten seien bereits im Datenbankpuffer vorhanden (d.h., sie brauchen die Kosten zum Navigieren im Baum nicht berücksichtigen). Geben Sie die I/O Kosten an, um die Einträge in sortierter Reihenfolge auszulesen, wenn

(*) in den Dateneinträgen des Baumes tatsächlich die Daten gespeichert sind, und der Baum somit Ballung (Clustering) ausnutzt. [2]

Zuerst muss von der Wurzel zum linkesten Blatt abgestiegen werden, danach kann das Sequence Set ausgelesen werden. Bei 60% Auslastung gibt es $450/0.6 = 750$ Blätter, d.h. 750 I/Os.

(**) in den Dateneinträgen des Baumes Tupel IDs gespeichert sind, die erst verfolgt werden müssen, um das Datum auszulesen. Der Baum besitzt keine Ballung (gehen Sie bei Ihren Berechnungen vom worst case aus). [3]

Im worst case führt jedes Blatt im Baum durch die Indirektion (die Tupel ID) und die fehlende Ballung zu einem Seitenzugriff. Die Anzahl an nötigen Dateneinträgen (Blättern) im Baum ist gleich der Anzahl an Tupel der Datei, also 4500. Ein Dateneintrag braucht 12 Byte (4 Byte für den Suchschlüssel, 8 Byte für die Tupel ID), jede Seite hat 500 Bytes Nutzdaten, also ist die Anzahl der Seiten für Blätter im B+ Baum $(4500 \cdot 12)/(500 \cdot 0.6) = 180$. In Summe sind somit diese 180 I/Os und 4500 I/Os zum eigentlichen Auslesen der Daten mittels Tupel ID, also 4680 I/Os notwendig.

Aufgabe 3: (15)

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

1. Betrachten Sie zwei Relationen $R(ABC)$ und $S(ABD)$. Dann liefert der Ausdruck $\pi_{AB}(S) \bowtie R$ ausschließlich Tupel, die auch in R enthalten sind. wahr ✕ falsch ◯
2. Betrachten Sie zwei Relationen $R(ABC)$ und $S(ABD)$. Dann git auf jeden Fall folgende Gleichheit: $\pi_{AB}(R \bowtie S) = \pi_{AB}(R) \cap \pi_{AB}(S)$. wahr ✕ falsch ◯
3. Die Historie $r_1(A)$, $r_2(B)$, $w_2(B)$, $w_1(A)$, $w_1(C)$, $w_2(C)$, c_2, c_1 ist zwar beim Zwei-Phasen Sperrprotokoll jedoch nicht beim strengen Zwei-Phasen Sperrprotokoll möglich. wahr ✕ falsch ◯
4. Bei verletzten Deferred Constraints treten Fehler erst am Ende der Transaktion auf. wahr ✕ falsch ◯
5. Eine Relation R sei an 5 Netzwerk-Knoten materialisiert mit den Gewichten 5, 9, 7, 3, und 5. Dann sind $Q_r(R) = 10$ und $Q_w(R) = 20$ gültige Lese- bzw. Schreib-Quoren. wahr ✕ falsch ◯
6. Nehmen Sie an, dass eine relationale Datenbank um das objektrelationale Feature “Geschachtelte Relationen” erweitert wurde. Dann lassen sich unter Umständen Anfragen bezüglich 1:n-Relationen effizienter auswerten als ohne dieses Feature. wahr ✕ falsch ◯
7. Falls das Rollback von Datenbankänderungen unterbrochen wird, muss das Datenbanksystem beim Wiederanlauf in der Lage sein, das Rollback abzuschließen. wahr ✕ falsch ◯
8. Betrachten Sie die Kostenformel $2 * b_R * (1 + I)$ mit $I = \lceil \log_{m-1}(\lceil b_R/m \rceil) \rceil$ für das externe Sortieren. Dabei steht b_R für die Anzahl der Seiten der Relation R und m für die Anzahl der verfügbaren Puffer-Seiten. wahr ✕ falsch ◯
9. Betrachten Sie zwei Relationen $R(\underline{A}B)$ mit 5000 Tupeln und $S(AC)$ mit 3000 Tupeln, wobei A ein Fremdschlüssel von der Relation S auf den Primärschlüssel A in R ist. Dann ergibt der Ausdruck $R \bowtie S$ exakt 3000 Tupel. wahr ✕ falsch ◯
10. Beim Zweiphasen-Commit-Protokoll kann der Koordinator mittels Timer-Überwachung verhindern, dass beim Absturz eines Agenten die Beendigung einer Transaktion blockiert wird. 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)

Die folgende Datenbankbeschreibung gilt für die Aufgaben 4 – 7:

Gegeben ist folgendes stark vereinfachtes Datenbankschema, in dem die Daten von Fußball-Vereinen und Spielern gespeichert werden.

verein (<u>vid</u>, name, kapitän: *spieler.sid*)
spieler (<u>sid</u>, name, gehalt, spieltIn: *verein.vid*)

Jeder Verein hat eine eindeutige ID *vid*, einen Namen *name* und einen Spieler als Kapitän *kapitän*.

Jeder Spieler hat eine eindeutige ID *sid*, einen Namen *name*, ein Gehalt *gehalt* und spielt in einem Verein *spieltIn*.

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

Aufgabe 4: (5)

Geben Sie die CREATE TABLE Statements mit allen nötigen Constraints für die Tabellen `verein` und `spieler` an. Beachten Sie eventuelle zyklische Abhängigkeiten zwischen den Fremdschlüsseln.

Geben Sie je ein INSERT Statement für die vorerst leeren Relationen `verein` und `spieler` an (d.h.: einen Verein und einen Spieler als Kapitän).

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.

```
CREATE TABLE verein (
   vid INTEGER PRIMARY KEY,
   name VARCHAR(30),
   kapitän INTEGER
);

CREATE TABLE spieler (
   sid INTEGER PRIMARY KEY,
   name VARCHAR(30),
   gehalt INTEGER,
   spieltIn INTEGER
);

ALTER TABLE verein
   ADD CONSTRAINT fk_kapitän
   FOREIGN KEY (kapitän)
   REFERENCES spieler(sid)
   DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE spieler
   ADD CONSTRAINT fk_verein
   FOREIGN KEY (spieltIn)
   REFERENCES verein(vid)
   DEFERRABLE INITIALLY DEFERRED;

INSERT INTO verein VALUES (1, 'FC Grashalm', 1);
INSERT INTO spieler VALUES (1, 'A. P.', 10000, 1);
COMMIT;

DROP TABLE verein CASCADE;
DROP TABLE spieler CASCADE;
```

Aufgabe 5: (9)

Definieren Sie mittels PL/pgSQL einen Trigger der bei einem UPDATE und INSERT in die `spieler` Tabelle das Gehalt um 40% kürzt.

Wenn das Gehalt negativ ist darf die Zeile in die Tabelle nicht eingefügt bzw. die Zeile der Tabelle nicht verändert werden.

Beispiel: Es wird der Spieler (1, „B. Leder“, 1000, 1) eingefügt, der Trigger schreibt jedoch (1, „B. Leder“, 600, 1) in die Tabelle *spieler*. Beim Versuch (1, „B. Leder“, -1000, 1) wird die Datenbank nicht verändert. Bei `UPDATE spieler SET gehalt = gehalt - 1000;` werden nur jene Spieler verändert deren `gehalt` nach dem Update nicht negativ ist.

```
CREATE FUNCTION fConvertSpieler()
RETURNS trigger AS $$
BEGIN
  IF (NEW.gehalt < 0) THEN
    RETURN NULL;
  ELSE
    NEW.gehalt = NEW.gehalt*0.6;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER convertSpieler
BEFORE UPDATE OR INSERT ON spieler
FOR EACH ROW EXECUTE PROCEDURE fConvertSpieler();
```

Aufgabe 6: (6)

Vervollständigen Sie die Java Methode, die eine `ArrayList` von Spielern als Input nimmt und diese Spieler in die Datenbank schreibt. Für den Zugriff auf die Elemente der `ArrayList` der Spieler können Sie folgende Vorlage verwenden:

```
for(Spieler sp : alSpieler) {
    // Zugriff mit sp.id (int), sp.name (String), sp.gehalt (int) und sp.spieltIn (int).
}
```

Verwenden Sie für die INSERT-Befehle unbedingt ein **prepared Statement**, das in der Schleife jeweils mit den Attribut-Werten des aktuellen Spielers gefüllt wird.

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

Gehen Sie davon aus, dass die Klasse auf eine Postgres Datenbank zugreift, die auf `server.euro.at` läuft. Der Datebankname ist `uefa`, verwenden Sie den Datenbankuser `exam` mit dem Passwort `dbs`.

Sie brauchen sich um keine Fehlerbehandlung zu kümmern.

```
public class Importer {
    public void import(ArrayList<Spieler> alSpieler) throws Exception {
        Class.forName('org.postgresql.Driver');
        Connection c = DriverManager.getConnection('jdbc:postgresql://server.euro.at/uefa',
            'exam',
            'dbs');
        PreparedStatement pStmt = conn.prepareStatement
            ("INSERT INTO spieler VALUES (?,?,?,?)");
        for(Spieler sp : alSpieler) {
            pStmt.setInt(1, sp.id);
            pStmt.setString(2, sp.name);
            pStmt.setInt(3, sp.gehalt);
            pStmt.setInt(4, sp.spieltIn);
            pStmt.executeUpdate();
        }
        conn.commit();
        pstmt.close();
        conn.close();
    }
}
```

Aufgabe 7: (10)

Geben Sie die *Vereine* (vid und name) und das *Durchschnittsgehalt* jener Vereine aus, deren Kapitäne **unter dem Vereins-Durschnitt** verdienen.

Bedingungen für diese SQL-Query:

Verwenden Sie ein SELECT-Statement mit GROUP BY und HAVING. Ein geschachteltes SELECT darf *ausschließlich* in der HAVING-Klausel verwendet werden!

```
SELECT v.vid, v.name, avg(s.gehalt) AS s_avg
FROM verein v, spieler s
WHERE s.spieltIn = v.vid
GROUP BY s.spieltIn, v.vid, v.name, v.kapitän
HAVING avg(s.gehalt) >
(SELECT k.gehalt FROM spieler k WHERE k.sid = v.kapitän);
```

Gesamtpunkte: 75