Gruppe **A**

Bitte tragen Sie **SOFORT** und **LESERLICH** Namen und Matrikelnr. ein, und legen Sie Ihren Studentenausweis bereit.

PRÜFUNG AUS		30.01.2018
◯ DATENMODELLIERUNG 2 (**184.790**)	◯ DATENBANKSYSTEME (**184.686**)	**GRUPPE A**
Matrikelnr.	Familienname	Vorname

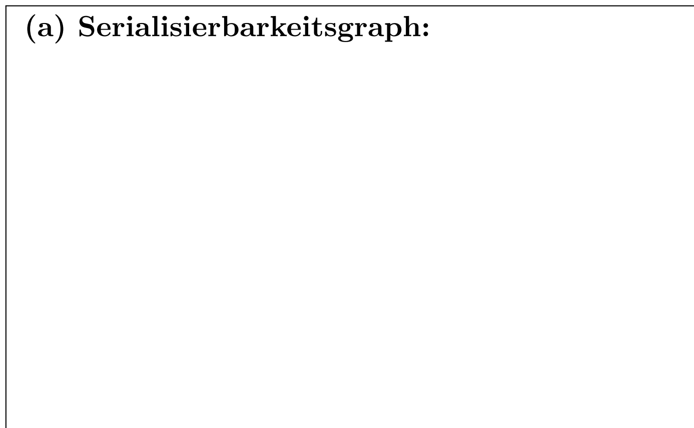
Arbeitszeit: 60 Minuten. Lösen Sie die Aufgaben auf den vorgesehenen Blättern; Lösungen auf Zusatzblättern werden nicht gewertet. **Viel Erfolg!**

Aufgabe 1: Eigenschaften von Transaktionen (7)

Betrachten Sie die unten angegebene Historie, welche durch eine Abfolge von Elementaroperationen der vier Transaktionen T_1, T_2, T_3 und T_4 auf den Datensätzen A, B, C und D gegeben ist. Dabei bedeutet $r_i(X)$ dass Transaktion T_i den Datensatz X liest (*read*), $w_i(X)$ dass Transaktion T_i den Datensatz X schreibt (*write*), und c_i steht für das `commit` der Transaktion T_i.

$$r_2(A), r_4(A), r_2(C), w_2(C), w_2(D), r_3(D), w_1(A), w_3(B), r_1(C), r_2(D), r_4(B), r_1(A), c_2, c_3, c_4, c_1$$

a) Zeichnen Sie den Serialisierbarkeitsgraphen der oben angegeben Historie.



b) Ist die Historie aus (a) konfliktserialisierbar? ◯ ja ◯ nein

Wenn ja, geben Sie bitte eine mögliche Reihenfolge an. Wenn nein: welche Transaktion(en) müssten entfernt werden, damit die verbleibende Historie konfliktserialisierbar wird?
(Achtung: Nur ankreuzen ohne einer Antwort auf den zweiten Teil gibt keinen Punkt!)

Aufgabe 2: Logging & Recovery (14)

Sie finden nach einem Crash Ihrer Datenbank (mit Verlust des Datenbankpuffers jedoch nicht des Hintergrundspeichers) die folgenden Log-Einträge vor. Im Hintergrundspeicher befinden sich die unten angegebenen Seiten.

Für "normale" Logeinträge wird das Format [LSN, TA, PageID, Redo, Undo, PrevLSN] verwendet, und für Kompensations Logeinträge (Compensation Log Records) wird das Format ⟨LSN, TA, PageID, Redo, UndoNextLSN, PrevLSN⟩ verwendet. `BOT` Log-Einträge verwenden das Format [LSN, TA, BOT, PrevLSN], und `COMMIT` Einträge das Format [LSN, TA, COMMIT, PrevLSN].

Beachten Sie, dass Undo/Redo-Einträge *relativ* zum Datenbestand mittels *Addition* bzw. *Subtraktion* angegeben werden, z.B.: [$\#i$, T_j, P_X, X+=d_1, X-=d_2, $\#k$] bedeutet, dass laut i-tem Eintrag die Transaktion T_j auf ein Datum X auf der Seite P_X schreibend zugreift, so dass beim Redo X um d_1 vergrößert werden müsste und beim Undo um d_2 verkleinert werden müsste und der vorangegangene Logeintrag dieser Transaktion die Nummer k hat.

Log-Einträge

[#1, T_3, BOT, 0]
[#2, T_1, BOT, 0]
[#3, T_2, BOT, 0]
[#4, T_1, P_A, A-=50, A+=50, #2]
[#5, T_2, P_A, A+=0, A-=0, #3]
[#6, T_3, P_C, C+=25, C-=25, #1]
[#7, T_2, P_B, B+=50, B-=50, #5]
[#8, T_1, P_C, C+=25, C-=25, #4]
[#9, T_3, P_B, B-=75, B+=75, #6]
[#10, T_2, P_C, C+=25, C-=25, #7]
[#11, T_3, COMMIT, #9]
⟨#12, T_2, P_C, C-=25, #7, #10⟩

Seiten im Hintergrundspeicher

P_A	LSN: #4
$A = 25$	

P_B	LSN: #0
$B = 120$	

P_C	LSN: #8
$C = 60$	

a) Bestimmen Sie die Werte für A, B und C nach der *Redo*-Phase.

A: .... B: .... C: ....

b) Führen Sie die *Undo*-Phase aus. Erzeugen Sie die Compensation Log Records (CLRs), und geben Sie die Werte für A, B und C nach Beendigung der *Undo*-Phase an. Schreiben Sie die CLRs auf die untenstehenden Zeilen; es kann sein, dass Sie nicht alle Zeilen benötigen. Verwenden Sie das oben beschriebene Format.

....

....

....

....

A: .... B: .... C: ....

Aufgabe 3: Locking/Sperrprotokolle (14)

(a) Gegeben ist die untenstehende Folge von Sperranforderungen und Freigaben von Sperren. Dabei bedeutet „$\text{lockS}_i(O)$“ (bzw. „$\text{lockX}_i(O)$“), dass die Transaktion T_i eine Lesesperre (bzw. eine Schreibsperre) auf das Datenobjekt O anfordert. Hingegen bedeutet „$\text{unlockS}_i(O)$“ (bzw. „$\text{unlockX}_i(O)$“), dass die Transaktion eine gehaltene Lesesperre (bzw. eine Schreibsperre) auf das Datenobjekt O freigibt.

$$\text{lockS}_2(B), \text{lockX}_4(D), \text{lockS}_1(B), \text{lockX}_3(A), \text{unlockX}_4(D), \text{lockS}_3(C), \text{lockX}_4(C)(*), \text{lockS}_4(B),$$
$$\text{lockX}_2(D), \text{unlockS}_1(B), \text{lockX}_1(A), \text{unlockX}_2(D), \text{unlockX}_1(A), \text{lockS}_2(A)(**)$$

Solange eine Transaktion blockiert ist (also eine Sperre angefordert aber noch nicht erhalten hat), überspringen Sie bitte Sperranforderungen bzw. Freigaben von Sperren dieser Transaktion (die Transaktion ist ja inaktiv). Wenn eine Transaktion nach einer Blockierung weiterlaufen kann, betrachten Sie bitte zuerst die übersprungenen Einträge dieser Transaktion, bis die Transaktion entweder wieder blockiert, oder bis alle übersprungenen Anfragen abgearbeitet wurden.

a) Welche der vier Transaktionen T_1, T_2, T_3 und T_4 verletzt das 2-Phasen Sperrprotokoll? Begründen Sie kurz Ihre Antwort.

Die folgenden Transaktionen verletzen das 2-Phasen Sperrprotokoll: ....

Begründung: ....

....

b) Skizzieren Sie bitte im ersten (linken) unten angegebenen Kästchen die Situation an der mit (∗) gekennzeichneten Stelle in der Folge, indem Sie die vorgegebene Tabelle vervollständigen. Tragen Sie in ein Feld ein X (bzw. ein S) ein, wenn die entsprechende Transaktion eine Schreibsperre (bzw. eine Lesesperre) auf dieses Datenobjekt besitzt. Tragen Sie für blockierte Transaktionen bitte zusätzlich für jene Sperranforderung auf Grund welcher die Transaktion nun blockiert ist (also jene, welche nicht gewährt wurde) ein WS (*wait shared*) bzw. WX (*wait exclusive*) in das entsprechende Feld ein.

c) Zeichnen Sie in das zweite (rechte) unten angegebene Feld den Wartegraphen für diese Transaktionen zum Zeitpunkt (∗∗).

(b) Aktuelle Sperren:

	A	B	C	D
T_1				
T_2				
T_3				
T_4				

(c) Wartegraph:

d) Herrscht zum Zeitpunkt (∗∗) ein Deadlock? ○ ja ○ nein

Falls ja, geben Sie bitte an, welche Transaktion(en) abgebrochen werden müssten um den Deadlock aufzulösen. Falls nicht, geben Sie eine zusätzliche Sperranforderung einer zum Zeitpunkt (∗∗) nicht blockierten Transaktion an, welche zum Zeitpunkt (∗∗) als nächste Anforderung sofort zu einem Deadlock führen würde.
(Achtung, ankreuzen alleine ohne eine Antwort auf den zweiten Teil gibt keine Punkte!)

....

Für die Aufgaben 4 – 6 gilt die Datenbankbeschreibung am letzten Blatt dieser Prüfung.

Aufgabe 4: CREATE Statements (11)

Die Tabelle `trainer` wurde bereits angelegt.

```
CREATE TABLE trainer(
   id INTEGER PRIMARY KEY,
   name VARCHAR(20) NOT NULL,
   buddy INTEGER,
   candies INTEGER
);
```

Geben Sie nun CREATE-Statements mit allen entsprechenden Constraints für die Tabellen `pokemon` und `caught` an. Die Spalte `evolution_candy` in `pokemon` hat als Standardwert 0. Schreiben Sie zusätzlich ein Statement, das die Tabelle `trainer` so ändert, dass der Fremdschlüssel der Spalte `buddy` richtig abgebildet wird.

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

Aufgabe 5: Rekursive Abfragen (15)

Geben Sie ein SELECT-Statement (ohne `WITH RECURSIVE`) an, das zweistufige Entwicklungen ausgibt.

Für die Beispielinstanz soll z.B. folgende Tabelle ausgegeben werden:

id	evolution_from	candy
76	74	125
95	807	37
208	808	62

Es kann zum Beispiel das Pokémon mit der id 76 mit Hilfe von 125 Candies aus dem Pokémon mit der id 74 entwickelt werden (aus 74 wird mit 25 Candies Pokémon 75 und aus 75 wird mit 100 Candies Pokémon 76).

Geben Sie nun eine rekursive Abfrage an, die alle möglichen Entwicklungen samt der Anzahl der benötigten Candies ausgibt.

Aufgabe 6: PL/SQL Trigger (9)

Nehmen Sie an, dass die Funktion und der Trigger wie am letzten Blatt dieser Prüfung definiert wurde.

Es wird nun folgendes UPDATE-Statement über die Beispielinstanz ausgeführt. Geben Sie die Ausgabe der SELECT-Statements für die Tabelle `trainer` und `caught` an.

```
UPDATE caught SET pid = 75 WHERE tid = 1 AND pid = 74;
```

```
SELECT * FROM trainer WHERE id = 1;    SELECT * FROM caught WHERE tid = 1;
```

trainer			
id	name	buddy	candies

caught		
tid	pid	cnt

Über die geänderte Instanz wird nun folgendes UPDATE-Statement ausgeführt. Geben Sie die Ausgabe der SELECT-Statements für die Tabelle `trainer` und `caught` an.

```
UPDATE caught SET pid = 208 WHERE tid = 1 AND pid = 95;
```

```
SELECT * FROM trainer WHERE id = 1;    SELECT * FROM caught WHERE tid = 1;
```

trainer			
id	name	buddy	candies

caught		
tid	pid	cnt

Über die geänderte Instanz wird nun folgendes UPDATE-Statement ausgeführt. Geben Sie die Ausgabe der SELECT-Statements für die Tabelle `trainer` und `caught` an.

```
UPDATE caught SET pid = 121 WHERE tid = 2 AND pid = 120;
```

```
SELECT * FROM trainer WHERE id = 2;    SELECT * FROM caught WHERE tid = 2;
```

trainer			
id	name	buddy	candies

caught		
tid	pid	cnt

Gesamtpunkte: 70

Sie können diese Seite abtrennen und brauchen ihn nicht abgeben!

Diesen Zettel daher bitte nicht beschriften! (Lösungen auf diesem Zettel werden nicht gewertet!)

Die folgende Datenbankbeschreibung gilt für die Aufgaben 4 – 6:

Gegeben ist folgendes stark vereinfachtes Datenbankschema zum Speichern von Pokémon und Trainern, die Pokémon sammeln.

trainer(id, name, buddy: *pokemon.id*, candies)
pokemon(id, name, evolution_from: *pokemon.id*, evolution_candy)
caught(tid: *trainer.id*, pid: *pokemon.id*, cnt)

Auf der Rückseite finden Sie eine Beispielinstanz dieses Schemas!

Jeder Trainer `trainer` hat eine eindeutige Identifikationsnummer `id` und einen Namen `name`. Weiters, verweist `buddy` auf das aktuelle Buddy-Pokémon und `candies` speichert die Anzahl der gesammelten Candies.

Jedes Pokémon `pokemon` hat eine eindeutige Identifikationsnummer `id` und einen Namen `name`. Außerdem kann ein Pokémon aus einem anderen Pokémon entwickelt werden. Dazu wird die `id` des Pokémons in `evolution_from` gespeichert aus dem das Pokémon entwickelt werden kann und die Anzahl der Candies `evolution_candy` die für die Entwicklung notwendig sind.

Die Anzahl der gefangenen Pokémon pro Trainer wird in der Tabelle `caught` gespeichert.

Hinweis: Nicht alle Pokémon in der Beispielinstanz kommen auch im Spiel vor ;).

Beispielinstanz für Aufgabe 4 – 6:

pokemon			
id	name	evolution_from	evolution_candy
74	'Kleinstein'	NULL	0
75	'Georock'	74	25
76	'Geowaz'	75	100
95	'Onix'	808	12
120	'Sterndu'	NULL	0
121	'Starmie'	120	50
208	'Stahlos'	95	50
807	'Munix'	NULL	0
808	'Minix'	807	25

trainer			
id	name	buddy	candies
1	'Rocko'	95	50
2	'Misty'	120	1000

caught		
tid	pid	cnt
1	74	1
1	95	1
2	120	2

Trigger für Aufgabe 6:

```
CREATE OR REPLACE FUNCTION fA() RETURNS TRIGGER AS $$
DECLARE
   v_c INTEGER;
BEGIN
   SELECT evolution_candy INTO v_c FROM pokemon WHERE id = NEW.pid AND evolution_from = OLD.pid;
   IF v_c IS NOT NULL THEN
      IF EXISTS(SELECT id FROM trainer WHERE id = NEW.tid AND candies > v_c) THEN
         IF (NEW.cnt>1) THEN
            IF EXISTS(SELECT * FROM caught WHERE tid = NEW.tid AND pid = NEW.pid) THEN
               UPDATE caught SET cnt = cnt + 1 WHERE tid = NEW.tid AND pid = NEW.pid;
            ELSE
               INSERT INTO caught VALUES (NEW.tid,NEW.pid,1);
            END IF;
            NEW.pid := OLD.pid;
            NEW.cnt := OLD.cnt - 1;
         END IF;
         UPDATE trainer SET candies = candies - v_c WHERE id = NEW.tid;
      ELSE
         RETURN NULL;
      END IF;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trA BEFORE UPDATE
   ON caught FOR EACH ROW EXECUTE PROCEDURE fA();
```