Gruppe **A**

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

PRÜFUNG AUS	**MUSTERLÖSUNG**	12.03.2019
◯ DATENMODELLIERUNG 2 (**184.790**)	◯ DATENBANKSYSTEME (**184.686**)	**GRUPPE A**

Matrikelnr.	Familienname	Vorname

Arbeitszeit: 90 Minuten. Lösen Sie die Aufgaben auf den vorgesehenen Blättern; Lösungen auf Zusatzblättern werden nicht gewertet. **Viel Erfolg!**

Aufgabe 1: Sperrprotokolle (13)

Auf der nächsten Seite ist eine Sequenz von Elementaroperationen der Transaktionen T_1, T_2, T_3 und T_4 auf den Datenbankobjekten A, B und C gegeben. *Dabei bezeichnen b_i und c_i den Beginn bzw. den erfolgreichen Abschluss (commit) von Transaktion T_i, und $r_i(O)$ und $w_i(O)$ Lese- bzw. Schreibzugriffe von Transaktion T_i auf das Objekt O. Die Einträge restart? können in Aufgabe a) ignoriert werden, ihre Bedeutung wird in Aufgabe b) beschrieben.*

a) Zur Kontrolle der Nebenläufigkeit dieser Zugriffe soll das *2-Phasen Sperrprotokoll* verwendet werden. Geben Sie die Folge von Sperranforderungen und Freigaben an, welche dabei durch die gegebene Historie erzeugt werden.

Zeichnen Sie außerdem den Wartegraphen zu dem mit (*) markierten Zeitpunkt.

Notation und Konventionen: Verwenden Sie $\mathrm{SL}_i(O)$ und $\mathrm{XL}_i(O)$ um anzuzeigen dass Transaktion T_i eine Lese- bzw. Schreibsperre auf Objekt O anfordert. Verwenden Sie $\mathrm{gSL}_i(O)$ und $\mathrm{gXL}_i(O)$ um zu notieren dass T_i eine angeforderte Lese- bzw. Schreibsperre erhalten hat ("granted"), und $wait_i$ um anzuzeigen dass Transaktion T_i angehalten wurde. Für Freigaben verwenden Sie bitte $\mathrm{rSL}_i(O)$ und $\mathrm{rXL}_i(O)$ ("release").

Nehmen Sie an, dass Sperren jeweils erst direkt vor dem Zugriff angefordert werden, und ass nur zwingend notwendige Sperren angefordert werden. Den Zeitpunkt der Freigaben können Sie – innerhalb der Einschränkungen durch das 2-Phasen Sperrprotokoll – frei wählen.

Wird eine Transaktion angehalten, so werden ihre weiteren Aktionen übersprungen (die Transaktion ist ja blockiert). Kann durch eine Freigabe eine blockierte Transaktion weiterlaufen, so werden alle übersprungenen Aktionen sofort nachgeholt. Warten mehrere Transaktionen auf die selbe Freigabe, so erhält jene Transaktion die Sperre, welche sie zuerst angefordert hat.

b) Um mögliche Deadlocks zu vermeiden soll nun das 2-Phasen Sperrprotokoll mit der *wound-wait Strategie* zur Deadlockvermeidung kombiniert werden.

Weisen Sie dazu jeder Transaktion einen korrekten Zeitstempel zu. Geben Sie außerdem für jede neu gestartete Transaktion ihren Zeitstempel nach dem Neustart an:

T_1: 1 .... T_2: 2 .... T_3: 3 .... T_4: 4 .... Neustart: T_3 : 3 .....

Geben Sie ebenfalls die Folge von Sperranforderungen an, welche bei Abarbeitung der Historie erzeugt werden, und zeichnen Sie den Wartegraphen zu dem mit (*) markierten Zeitpunkt.

Notation und Konventionen: Befolgen Sie die in a) beschriebenen Anweisungen. Verwenden Sie zusätzlich $reset_i$ um anzuzeigen, dass Transaktion T_i auf Grund des Protokolls zurückgesetzt wurde. Bei den Einträgen *restart?* soll jene zurückgesetzte (und noch nicht neu gestartete) Transaktion neu gestartet werden deren zurücksetzen am längsten zurück liegt.

T_1	T_2	T_3	T_4
b_1			
$r_1(B)$			
	b_2		
$w_1(C)$			
	$r_2(B)$		
		b_3	
		$r_3(A)$	
$w_1(A)$			
		$r_3(B)$	
	$w_2(A)$		
	$r_2(B)$		
	$w_2(A)$		
			b_4
		$w_3(C)$	
			$w_4(A)$
restart?			
(*)			
$w_1(A)$			
	c_2		
c_1			
restart?			
			c_4
restart?			
		c_3	

Sperren für Aufgabe a)

$\mathrm{SL}_1(B), \mathrm{gSL}_1(B), \mathrm{XL}_1(C), \mathrm{gXL}_1(C), \mathrm{SL}_2(B), \mathrm{gSL}_2(B),$ .....

$\mathrm{SL}_3(A), \mathrm{gSL}_3(A), \mathrm{XL}_1(A), \mathit{wait}_1, \mathrm{SL}_3(B), \mathrm{gSL}_3(B),$ .....

$\mathrm{XL}_2(A), \mathit{wait}_2, \mathrm{XL}_3(C), \mathit{wait}_3, \mathrm{XL}_4(A), \mathit{wait}_4$ .....

.....

.....

Sperren für Aufgabe b)

$\mathrm{SL}_1(B), \mathrm{gSL}_1(B), \mathrm{XL}_1(C), \mathrm{gXL}_1(C), \mathrm{SL}_2(B), \mathrm{gSL}_2(B),$ .....

$\mathrm{SL}_3(A), \mathrm{gSL}_3(A), \mathrm{XL}_1(A), \mathit{reset}_3, \mathrm{rSL}_3(A), \mathrm{gXL}_1(A),$ .....

$\mathrm{XL}_2(A), \mathit{wait}_2, \mathrm{XL}_4(A), \mathit{wait}_4, \mathrm{SL}_3(A), \mathit{wait}_3,$ .....

$\mathrm{rXL}_1(C), \mathrm{rSL}_1(B), \mathrm{rXL}_1(A),$ .....

$\mathrm{gXL}_2(A), \mathrm{rSL}_2(B), \mathrm{rXL}_2(A),$ .....

$\mathrm{gXL}_4(A), \mathrm{rXL}_4(A), \mathrm{gSL}_3(A),$ .....

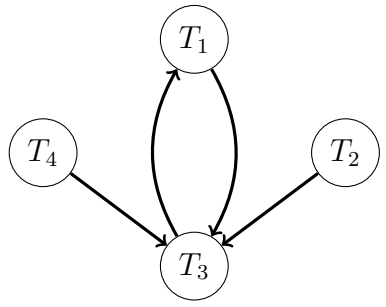
$\mathrm{SL}_3(B), \mathrm{gSL}_3(B), \mathrm{XL}_3(C), \mathrm{gXL}_3(C),$ .....

$\mathrm{rSL}_3(A), \mathrm{rSL}_3(B), \mathrm{rXL}_3(C)$ .....

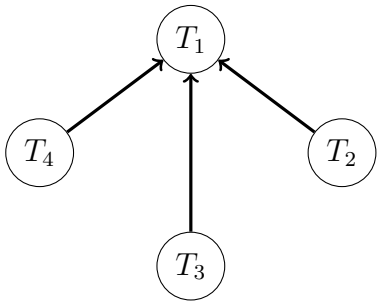
.....

.....

Wartegraph für Aufgabe a)



Wartegraph für Aufgabe b)



Aufgabe 2: Logging & Recovery (13)

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.

Wir verwenden die selbe Notation wie in der Vorlesung: "Normale" Logeinträge haben das Format [LSN, TA, PageID, Redo, Undo, PrevLSN], und Kompensations Logeinträge (Compensation Log Records) haben das Format ⟨LSN, TA, PageID, Redo, PrevLSN, UndoNextLSN⟩. `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_2, `BOT`, #0]
[#2, T_1, `BOT`, #0]
[#3, T_2, P_D, D+=20, D-=20, #1]
[#4, T_4, `BOT`, #0]
[#5, T_2, P_A, A-=5, A+=5, #3]
[#6, T_1, P_A, A+=10, A-=10, #2]
[#7, T_3, `BOT`, #0]
[#8, T_1, P_D, D-=5, D+=5, #6]
[#9, T_3, P_C, C+=5, C-=5, #7]
[#10, T_1, P_C, C+=5, C-=5, #8]
[#11, T_3, P_A, A-=5, A+=5, #9]
[#12, T_4, P_D, D+=15, D-=15, #4]
⟨#13, T_1, P_C, C-=5, #10, #8⟩
[#14, T_4, P_A, B-=5, B+=5, #12]
[#15, T_2, P_C, C+=25, C-=25, #5]
⟨#16, T_1, P_D, D+=5, #13, #6⟩
⟨#17, T_1, P_A, A-=10, #16, #2⟩
[#18, T_2, `COMMIT`, #15]
⟨#19, T_1, `BOT`, #17⟩
[#20, T_4, P_A, A-=20, A+=20, #14]
[#21, T_3, P_D, D+=10, D-=10, #11]

Seiten im Hintergrundspeicher

P_A	LSN: #11
$A = 20$	$B = 10$

P_C	LSN: #13
$C = 30$	

P_D	LSN: #8
$D = 50$	

Es wird nun ein Recovery nach dem ARIES Verfahren durchgeführt.

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

A: `-10` ..... B: `5` ..... C: `55` ..... D: `80` .....

b) Führen Sie die *Undo*-Phase aus. Erzeugen Sie die Compensation Log Records (CLRs), und geben Sie die Werte für A, B, C und D 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.

⟨#22, T_3, P_D, D-=10, #21, #11⟩ .....
⟨#23, T_4, P_A, A+=20, #20, #14⟩ .....
⟨#24, T_4, P_A, B+=5, #23, #12⟩ .....
⟨#25, T_4, P_D, D-=15, #24, #4⟩ .....
.....
⟨#26, T_3, P_A, A+=5, #22, #9⟩ .....
⟨#27, T_3, P_C, C-=5, #26, #7⟩ .....
⟨#28, T_3, BOT, #27⟩ .....
⟨#29, T_4, BOT, #25⟩ .....
.....

A: `15` ..... B: `10` ..... C: `50` ..... D: `55` .....

c) Nehmen Sie an, nach dem Logeintrag mit der LSN #20 wäre ein aktionskonsistenter Sicherungspunkt angelegt worden. Bei welchem Logeintrag (LSN) könnte in so einem Fall das globales Redo beginnen, und bis zu welchem Logeintrag müsste man beim globalen Undo maximal zurückgehen?

Redo: #21 ..... *Undo*: #4 .....

Aufgabe 3: Eigenschaften von Transaktionen (9)

a) 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. *Die Bedeutung der Operationen ist wie in Aufgabe 1. Die Operation a_i steht für den Abbruch von Transaktion T_i.*

Geben Sie an, zwischen welchen Transaktionen eine Leseabhängigkeit besteht. D.h., geben Sie für jede Transaktion an, von welchen anderen Transaktionen diese Transaktion liest (falls eine Transaktion von keiner anderen Transaktion liest, streichen Sie das entsprechende Feld bitte durch).

Bestimmen Sie anschließend, ob die Historie rücksetzbar ist oder nicht. Geben Sie eine kurze Begründung an Hand der Historie, an.
(Achtung, ankreuzen alleine ohne eine Begründung gibt keine Punkte!)

T_1	T_2	T_3	T_4
b_1	b_2	b_3	b_4
		$r_3(A)$	
	$r_2(B)$		
			$r_4(C)$
			$w_4(B)$
$r_1(A)$			
		$w_3(A)$	
			$w_4(C)$
$w_1(A)$			
	$w_2(B)$		
			$r_4(B)$
		$r_3(C)$	
	$w_2(C)$		
a_1			
			$w_4(B)$
	$r_2(C)$		
	$r_2(A)$		
		$r_3(B)$	
	c_2		
		c_3	
			c_4

Leseabhängigkeiten:

T_1 liest von ..... T_2 liest von T_3 .....

T_3 liest von T_4 ..... T_4 liest von T_2 .....

Historie ist Rücksetzbar:

◯ ja ⊗ nein

Begründung: Transaktion T_3 liest von Transaktion T_4, hat ihr COMMIT jedoch vor dem ..... COMMIT von Transaktion T_4. Würde T_4 abbreachen, könnte T_3 nicht mehr zurückgesetzt werden. .....

.....

b) Geben Sie eine Historie an welche kaskadierendes Rücksetzen vermeidet, jedoch nicht strikt ist. Verwendet Sie dazu maximal drei Transaktionen T_1, T_2 und T_3, sowie drei Datenbankobjekte A, B und C. Geben Sie neben Lese- und Schreiboperationen $r_i(O)$ und $w_i(O)$ auch den Beginn b_i und das Ende c_i oder a_i der Transaktionen an.

b)

$$b_1, b_2, w_1(A), w_2(A), c_1, c_2$$

Für die Aufgaben 4 – 6 gilt die Datenbankbeschreibung auf diesem Blatt.

Aufgabe 4: Erstellen eines Datenbankschemas mittels SQL (7)

Folgendes Schema sei gegeben:

hafen(ort, land, aktiv)
schiff(id, ladung, ladung_art, max_ladung, heimat_ort: *hafen.ort*, heimat_land: *hafen.land*)
liegt_an(sid: *schiff.id*, entladen, hafen_ort: *hafen.ort*, hafen_land: *hafen.land*, woche)

Ein Hafen besteht aus einem eindeutigen Paar von Ort und Land. Ein Schiff hat eine eindeutige ID, die (momentane) Ladung, die maximale erlaubte Ladung und eine Ladungsart. Die Ladungsart ist ein Enum bestehend aus einem von vier Werten: 'Oel', 'Container', 'Fisch' oder 'Leer'. Zusätzlich wird für jedes Schiff sein Heimathafen gespeichert.

Jedes Schiff kann an Häfen anlegen. Es wird vermerkt in welcher Kalenderwoche es angelegt hat. Es wird angenommen, dass ein Schiff immer nur eine volle Kalenderwoche anlegen kann, die Kombination bestehend aus der ID des Schiffes und der Woche muss also einzigartig sein. Für jedes Anlegen wird vermerkt, wie viel Ladung umgeschlagen wird. Zuletzt soll auch sichergestellt werden, dass die Kalenderwoche größer 0 ist.

Geben Sie die nötigen SQL Statements an, um obiges Schema (inklusive aller Konsistenzbedingungen) anzulegen. Sie können dabei entsprechende (einfache) Datentypen für die Attribute wählen.

```
CREATE TABLE hafen (
   ort VARCHAR(100),
   land VARCHAR(100),
   aktiv BOOLEAN NOT NULL,
   PRIMARY KEY(ort, land)
);

create type ladungstyp as enum('Oel', 'Container', 'Fisch', 'Leer');


CREATE TABLE schiff (
   id INTEGER PRIMARY KEY,
   ladung INTEGER NOT NULL,
   ladung_art ladungstyp NOT NULL,
   max_ladung INTEGER NOT NULL,
   heimat_ort VARCHAR(100) NOT NULL,
   heimat_land  VARCHAR(100) NOT NULL,
   FOREIGN KEY(heimat_ort,heimat_land) REFERENCES hafen(ort,land)
);


CREATE TABLE liegt_an(
  sid INTEGER NOT NULL,
  entladen INTEGER NOT NULL,
  hafen_ort VARCHAR(100) NOT NULL,
  hafen_land VARCHAR(100) NOT NULL,
  woche INTEGER NOT NULL CHECK (woche > 0),
  FOREIGN KEY(hafen_ort,hafen_land) REFERENCES hafen(ort,land),
  FOREIGN KEY(sid) REFERENCES schiff(id),
  PRIMARY KEY(sid, woche)
);
```

Hinweis: Achten Sie bei den Statements auf die Reihenfolge.

Aufgabe 5: Views und Rekursive Abfragen (14)

a) Erstellen Sie eine View, basierend auf dem Schema der letzten Aufgabe.

Es ist gefragt, dass diese View

1) Schiffs-IDs, Hafen-Orte, Hafen-Länder und Wochen anzeigt,
2) die aus der Relation "liegt_an" stammen und,
3) bei denen der betreffende Hafen "aktiv" ist, also den Wert 'true' in dieser Spalte hat.

Geben Sie die nötigen SQL Statements an, um eine View anzulegen, die obige Bedingungen erfülllt.

```
CREATE VIEW liegt_an_aktiv AS
(
SELECT  sid, ort,land,woche
FROM    hafen JOIN liegt_an
        ON (hafen.ort = liegt_an.hafen_ort
        AND hafen.land = liegt_an.hafen_land)
WHERE   aktiv = 'true'
);
```

b) Erstellen Sie eine Rekursive SQL Abfrage

Realisieren Sie, basierend auf dem Schema der letzten Aufgabe, folgende Abfrage:

Es sei jedes Paar an Schiffen, die zur selben Kalenderwoche am selben *aktiven* Hafen anlegen, als *Schiffsnachbarn* von einander definiert.

Geben Sie die transitive Hülle der Schiffsnachbarn des Schiffes mit ID '1' an. D.h. : Alle Schiffe die Schiffsnachbarn des Schiffs mit ID '1' sind, alle Schiffe die Schiffsnachbarn von Schiffsnachbarn des Schiffes mit ID '1' sind, so wie deren Schiffsnachbarn und so weiter.

Geben Sie die nötigen SQL Statements an, um eine Rekursive Abfrage zu schreiben welche die beschriebene Abfrage implementiert.

Sie dürfen die View aus dem vorherigen Beispiel hier wiederverwenden!

```
WITH RECURSIVE Schiffsnachbarn(sid) AS
(
SELECT  t2.sid
FROM    liegt_an_aktiv t1 JOIN liegt_an_aktiv t2
        ON (t1.ort = t2.ort AND t1.land = t2.land
        AND t1.woche = t2.woche )
WHERE   t1.sid = '1'
UNION
SELECT  t2.sid
FROM    (Schiffsnachbarn NATURAL JOIN liegt_an_aktiv) t1
        JOIN liegt_an_aktiv t2 ON (t1.ort = t2.ort
        AND t1.land = t2.land AND t1.woche = t2.woche)
)
SELECT * FROM Schiffsnachbarn;
```

Hinweis: Achten Sie darauf, dass ihre Abfrage terminiert.

Aufgabe 6: PL/SQL Trigger (14)

a) Nehmen Sie an, dass die **Funktion `fLAU` und der Trigger `trLAU`** wie am letzten Blatt dieser Prüfung definiert wurden.

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

```
UPDATE liegt_an SET entladen = 500 WHERE woche = 17 AND sid = 5;

SELECT * FROM schiff WHERE id = 5;
SELECT * FROM liegt_an WHERE woche = 17;
```

```
schiff: (5 , 400 , Fisch , 800 , Wien , AT)
liegt_an: (5 , 500 , Innsbruck , AT , 17)
```

Es wird nun folgendes UPDATE Statement **über die Beispielinstanz** ausgeführt. Geben Sie die Ausgabe der SELECT-Statements für die Tabelle schiff und liegt_an an.

```
UPDATE liegt_an SET sid = 3, entladen = 100 WHERE sid = 4;

SELECT * FROM schiff WHERE id=3 OR id=4;
SELECT * FROM liegt_an WHERE sid=3 OR sid=4;
```

schiff: (3 , 0 , Leer , 100 , Monrovia , LR) (4 , 300 , Container , 300 , Nassau , BS)
liegt_an: (3 , 100 , Monrovia , LR , 1), (3 , 100 , Wien , AT , 12)

Es wird nun folgende Statements **über die Beispielinstanz** ausgeführt. Geben Sie die Ausgabe der SELECT-Statements für die Tabelle schiff und liegt_an an.

```
  BEGIN;
UPDATE liegt_an SET sid = 2, entladen = 100, hafen_ort = 'Las Vegas' WHERE woche = 5;
UPDATE liegt_an SET sid = 1, entladen = 200 WHERE woche = 5;
COMMIT;

SELECT * FROM schiff WHERE id <= 2;
SELECT * FROM liegt_an WHERE woche=5;
```

schiff: (1 , 150 , Oel , 300 , Panama , PA), (2 , 500 , Oel , 900 , Monrovia , LR)
liegt_an: (1 , 100 , Panama , PA , 5)

---

b) Nehmen Sie an, dass *zusätzlich* zu fLAU und trLAU nun auch die Funktion fUNL und der Trigger trUNL wie am letzten Blatt dieser Prüfung definiert wurde.

Es wird nun folgendes UPDATE Statement **über die Beispielinstanz** ausgeführt. Beschreiben Sie was passiert. Dokumentieren Sie alle Einträge die sich in der Datenbank ändern.

```
UPDATE schiff SET ladung = 200 WHERE id = 4;
```

Die beiden Trigger rufen sich in einer Endlosschleife gegenseitig auf.

Sie können diesen Zettel abtrennen und brauchen ihn nicht abgeben!

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

Beispielinstanz für Aufgabe 6:

Schiff:

id	ladung	ladung_art	max_ladung	heimat_ort	heimat_land
1	150	Oel	300	Panama	PA
2	500	Oel	900	Monrovia	LR
3	100	Fisch	100	Monrovia	LR
4	200	Container	300	Nassau	BS
5	700	Fisch	800	Wien	AT

Hafen:

ort	land	aktiv
Wien	AT	true
Innsbruck	AT	true
Nassau	BS	true
Monrovia	LR	true
Panama	PA	true

liegt_an:

sid	entladen	hafen_ort	hafen_land	woche
1	100	Panama	PA	5
5	100	Panama	PA	8
5	200	Innsbruck	AT	17
3	100	Monrovia	LR	1
4	100	Wien	AT	12

Trigger für Aufgabe 6a.

```
CREATE FUNCTION fLAU() RETURNS TRIGGER AS $$
DECLARE
  nl INTEGER;
BEGIN
  IF EXISTS(SELECT * FROM schiff WHERE id = NEW.sid) AND
     EXISTS(SELECT * FROM schiff WHERE id = OLD.sid) THEN
       UPDATE schiff SET ladung = ladung + OLD.entladen WHERE id = OLD.sid;
       UPDATE schiff SET ladung = ladung - NEW.entladen WHERE id = NEW.sid;

       IF EXISTS (SELECT * FROM schiff WHERE id = OLD.sid AND ladung > max_ladung)
       THEN
         DELETE FROM schiff WHERE id = OLD.sid;
       END IF;

       SELECT ladung INTO nl FROM schiff WHERE id = NEW.sid;
       IF nl < 0 THEN
         RAISE EXCEPTION 'Entladung unmoeglich';
       ELSIF nl = 0 THEN
         UPDATE schiff SET ladung_art = 'Leer' WHERE id = NEW.sid;
       END IF;
  ELSE
    RAISE EXCEPTION 'Fehlende Schiffe';
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trLAU BEFORE UPDATE ON liegt_an
       FOR EACH ROW EXECUTE PROCEDURE fLAU();
```

Zusätzlicher Trigger für Aufgabe 6b.

```
CREATE FUNCTION fUNL() RETURNS TRIGGER AS $$
DECLARE
  last INTEGER;
BEGIN
  SELECT MAX(woche) INTO STRICT last FROM liegt_an WHERE sid = NEW.id;
  UPDATE liegt_an SET entladen = entladen + NEW.ladung - OLD.ladung
    WHERE sid = NEW.id AND woche = last;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trUNL BEFORE UPDATE OF ladung ON schiff
       FOR EACH ROW EXECUTE PROCEDURE fUNL();
```