Übung Datenbanksysteme II

Recovery

Tobias Bleifuß

Folien basierend auf Folien von Leon Bornemann, Maximilian Jenders und Thorsten Papenbrock

Agenda

1. Nachbesprechung Hausaufgabe 5
2. Recovery
3. Undo-Logging
4. Redo-Logging
5. Undo/Redo-Logging
6. Aufgaben
7. Hausaufgabe 6

Nachbesprechung Hausaufgabe 5

Aufgabe 1: Kardinalitätsschätzung

Gegeben seien die folgenden Relationen und deren Statistiken:

$W(a,b)$	$X(b,c)$	$Y(c,d)$	$Z(d,a)$
$T(W) = 300$	$T(X) = 600$	$T(Y) = 900$	$T(Z) = 1200$
$V(W,a) = 30$			$V(Z,a) = 300$
$V(W,b) = 60$	$V(X,b) = 50$		
	$V(X,c) = 100$	$V(Y,c) = 50$	
		$V(Y,d) = 60$	$V(Z,d) = 40$

Schätze die Kardinalität der

a) $W \bowtie X \bowtie Y \bowtie Z$
b) $\sigma_{a=10}(W)$
c) $\sigma_{c=20}(Y)$
d) $\sigma_{c=20}(Y) \bowtie Z$
e) $W \times Y$
f) $\sigma_{d>10}(Z)$
g) $\sigma_{a=1 \wedge d=2}(Z)$
h) $\sigma_{c>1 \wedge d=2}(Y)$
i) $X \overset{\bowtie}{}_{X.b=Z.d} Z$

Problem:
Selektionsergebnis: 18 Tupel
V(Y,d) = 60
Man rechnet trotzdem mit 60 weiter, da die maximal 18 verschiedenen Werte aus 60 ausgewählt wurden. Wieso ergibt das Sinn?

→ In Wahrscheinlichkeiten denken ist intuitiv! Die Wahrscheinlichkeit, dass ein Tupel aus Z zu einem Tupel aus dem Resultat der Selektion passt ist weiterhin 1/60

Nachbesprechung Hausaufgabe 5

Aufgabe 2: Join-Kardinalität

Gegeben sind zwei Relationen $R(A, B)$ und $S(B, C)$. Beide Relationen enthalten 20 unterschiedliche Werte im Attribut B, wobei die Werte in R den Werten in S entsprechen. Es gilt also $V(R, B) = V(S, B) = 20$. Die Werteverteilungen in $R(B)$ und $S(B)$ sind durch folgendes Histogramm beschrieben, welches die Häufigkeit der 4 häufigsten Werte angibt:

	0	1	2	3	4	andere Werte
$R.B$	5	6	4	5	*	32
$S.B$	10	8	5	*	7	48

Die mit * gekennzeichneten Werte gehören nicht zu den vier jeweils häufigsten Werten, sondern zu den "anderen Werten". Schätze nun unter Verwendung des Histogramms die Kardinalität des Joins über $R(A, B) \bowtie S(B, C)$ ab. Schätze anschließend die Kardinalität ohne das Histogram zu verwenden (wie üblich Annahme der Gleichverteilung aller 20 Attributwerte). Vergleiche die beiden Ergebnisse. **5 P**

Nachbesprechung Hausaufgabe 5

Aufgabe 3

- Meist richtig, eine Stolperfalle
- Beispiel:
  - X(a,b,c), Y(a,b), Z(c,d)
  - $T(X) = T(Y) = T(Z) = 1000$
  - $V(X, a) = V(X, b) = V(Y, a) = V(Y, b) = 100$
  - $V(X, c) = 1000, \ \mathrm{V(Z, c)} = 1$
  - $\mathrm{T(X} \bowtie Y) = \frac{1000 * 1000}{100 * 100} = 100$ (geschätzt)
  - $\mathrm{V(X} \bowtie Y, c) = 1000$ (preservation of value sets)
  - $\mathrm{T}\big((\mathrm{X} \bowtie Y) \bowtie Z\big) = \frac{100 * 1000}{1000}$

Trotzdem durch 1000 teilen! Warum?

Recovery

Fehlerarten:

- Fehlerhafte Dateneingabe
- Medienfehler
- Katastrophe
- Systemfehler

Recovery

Transaktionen

- Atomar
- Haben Zustand
  - Variablenwerte
  - Aktuelles Statement

Datenbank

- Zustand für jedes Element
  - Je nach Granularität: Tupel, Block, etc…
- Konsistenz
  - Explizite und implizite Nebenbedingungen sind erfüllt

Problem: Systemfehler kann jederzeit auftreten → Atomarität gefährdet

Lösung: Logging

Recovery

Operationen

- Transaktion
  - Read(X,t)
  - Write(X,t)
- Puffermanager
  - Input(X)
  - Output(X)

Undo-Logging

Log-Einträge

- <Start T>
- <Commit T>
- <Abort T>
- <T,X,v> (v: alter Wert)

Es gibt keine Änderungen in der Datenbank (auf Disk), die nicht im Log auftauchen!

Regeln:

- U1: Falls T Element X verändert muss <T,X,v> im log geschrieben werden bevor X auf Disk geschrieben wird
- U2: <Commit T> wird erst in das Log geschrieben nachdem alles auf Disk geschrieben wurde
- Recovery: Mache alle Transaktionen T rückgängig, bei denen <Commit T> nicht existiert.

Redo-Logging

Log-Einträge

- <Start T>
- <Commit T>
- <Abort T>
- <T,X,v> (v: neuer Wert)

Transaktionen ohne <commit T> im Log haben die Datenbank nicht verändert!

Regeln:

- R1: Bevor ein Wert in die Datenbank geschrieben wird müssen alle Einträge im Log sein, inklusive <Commit T>

Recovery: Führe alle Operationen der Transaktionen T erneut aus, bei denen <Commit T> existiert. Ignoriere den Rest

Undo/Redo-Logging

Nachteil Undo:

- Daten müssen direkt nach der Transaktion geschrieben werden
  → vorher kann die Anwendung kein „Successful" erhalten

Nachteil Redo:

- Alle veränderten Blöcke müssen im Puffer bleiben bis commit und Log-Datensätze auf Disk sind → hoher Speicherbedarf

Undo/Redo Logging:

- Schreibe nun <T,X,v,w> (v: alt, w: neu)
- Regel UR1: Falls T Element X verändert muss <T,X,v,w> im Log geschrieben werden bevor X auf Disk geschrieben wird
- Recovery: Undo für alle uncommitteten Transaktionen, Redo für alle committeten Transaktionen

Wie bei Undo/Redo

Keine Regel über commit
→ DBMS ist flexibel

Aufgabe 1: Schreib-Reihenfolgen

Relevante Schreib-Operationen auf Festplatte:

- Schreiben der Daten
- Schreiben der Change-Logs
- Schreiben des Commit-Logs

In welcher Reihenfolge werden die Operationen ausgeführt bei:

UNDO

1. Change
2. Daten
3. Commit

REDO

1. Change
2. Commit
3. Daten

UNDO/REDO

1. Change
2. Daten / Commit

Aufgabe 2: Recovery-Strategie

Wie erfolgt die Wiederherstellung eines konsistenten Zustands bei:

UNDO

1. Change
2. Daten
3. Commit

REDO

1. Change
2. Commit
3. Daten

UNDO/REDO

1. Change
2. Daten / Commit

- Log rückwärts durchlaufen:
  - Merke alle Transaktionen mit COMMIT oder ABORT
  - Bei Update-Eintrag <T,X,v>:
    - Falls für T COMMIT oder ABORT bekannt:
      - Ignoriere Eintrag
    - Sonst:
      - Schreibe v auf X und <ABORT T>
- Flush Log

Aufgabe 2: Recovery-Strategie

Wie erfolgt die Wiederherstellung eines konsistenten Zustands bei:

UNDO

1. Change
2. Daten
3. Commit

REDO

1. Change
2. Commit
3. Daten

UNDO/REDO

1. Change
2. Daten / Commit

- Identifiziere alle committeten Transaktionen
- Log vorwärts durchlaufen:
  - Bei Update-Eintrag <T,X,v`>:
    - Falls für T COMMIT bekannt:
      - Schreibe v` auf X
    - Sonst:
      - Ignoriere Eintrag
- Schreibe <ABORT T> für jede uncommittete Transaktion
- Flush Log

Aufgabe 2: Recovery-Strategie

Wie erfolgt die Wiederherstellung eines konsistenten Zustands bei:

UNDO

1. Change
2. Daten
3. Commit

REDO

1. Change
2. Commit
3. Daten

UNDO/REDO

1. Change
2. Daten / Commit

- REDO für alle committeten Transaktionen in chronologischer Reihenfolge
- UNDO für alle uncommitteten Transaktionen in umgekehrt chronologischer Reihenfolge
- Flush Log

Aufgabe 3:
Logging Optimierung

Kann beim Logging ohne Performance-Verluste auf <ABORT T>-Log-Einträge verzichtet werden bei:

UNDO	REDO	UNDO/REDO
Nein! Abgebrochene Transaktionen würden unnötig erneut zurück-gerollt werden.	Ja! Beim Recovery sind nur die committeten Transaktionen relevant.	Nein! Abgebrochene Transaktionen würden unnötig erneut zurück-gerollt werden.

Aufgabe 4: Log-Analyse

Gegeben:

- DB-Elemente A=0 und B=0
- Transaktion T, die A und B auf 1 ändert

Logfile:

- <START T>
- <T,A,0>
- <T,?,?>
- <COMMIT T>

In <T,A,0> steht der alte Wert von A
→ UNDO-Log

Fragen:

- Ist dies ein UNDO- oder REDO-Log?
- Welche Werte müssen die "?" enthalten?

Aufgabe 4: Log-Analyse

Gegeben:

- DB-Elemente A=0 und B=0
- Transaktion T, die A und B auf 1 ändert

Logfile:

- <START T>
- <T,A,0>
- <T,B,0>
- <COMMIT T>

In <T,A,0> steht der alte Wert von A
→ UNDO-Log

Fragen:

- Ist dies ein UNDO- oder REDO-Log?
- Welche Werte müssen die "?" enthalten?

Aufgabe 5:
Non-blocking Checkpointing

UNDO-Checkpointing:

- Schreibe Log-Eintrag <START CKPT (T1, …, Tk)>
  - T1, …, Tk sind alle aktiven Transaktionen
- Flush-Log
- Warte auf <COMMIT Ti> bzw. <ABORT Ti> aller T1, …, Tk
  - Erlaube dabei neue Transaktionen!
- Schreibe Log-Eintrag <END CKPT>
- Flush-Log

UNDO-Recovery:

?

Aufgabe 5:
Non-blocking Checkpointing

UNDO-Recovery:

Lese das Log rückwärts:

a. Lese zuerst <END CKPT>:
   - UNDO aller Transaktionen Ti ohne <COMMIT Ti>
     bis zum ersten <START CKPT>

b. Lese zuerst <START CKPT (T1, …, Tk)>:
   - UNDO aller Transaktionen Ti ohne <COMMIT Ti>
     bis zum letzten <START Tj> der Transaktionen T1, …,Tk

Aufgabe 5:
Non-blocking Checkpointing

REDO-Checkpointing:

- Schreibe Log-Eintrag <START CKPT (T1, …, Tk)>
  - T1, …, Tk sind alle aktiven Transaktionen
  - Aktiv: Commit noch nicht im Log!
- Flush-Log
- Warte bis alle Transaktionen Ti, die geänderte Daten im Puffer und ein <COMMIT Ti> vor dem Checkpoint haben, auf Disk geschrieben wurden
  - Erlaube dabei neue Transaktionen!
- Schreibe Log-Eintrag <END CKPT>
- Flush-Log

REDO-Recovery:

?

Aufgabe 5:
Non-blocking Checkpointing

REDO-Recovery:

Finde letzten Checkpoint-Datensatz im Log:

a. <END CKPT>:
   - Chronologisches REDO aller Transaktionen Ti mit <COMMIT Ti>, die im vorhergegangenen <START CKPT (T1, …, Tk)> stehen oder danach gestartet wurden; zurückgehen also bis zum letzten <START Ti> der T1, …, Tk

b. <START CKPT (T1, …, Tk)>:
   - Suche den vorherigen <END CKPT> und nutze Vorgehen aus a), da dieser Log-Eintrag nicht bei der Recovery hilft

Aufgabe 5:
Non-blocking Checkpointing

UNDO/REDO-Checkpointing:

- Schreibe Log-Eintrag <START CKPT (T1, …, Tk)>
  - T1, …, Tk sind alle aktiven Transaktionen
- Flush-Log
- Warte bis alle Aktionen aller Transaktionen, die zu diesem Zeitpunkt geänderte Daten im Puffer haben, eben diese Daten auf Disk geschrieben haben
  - Erlaube dabei neue Transaktionen!
- Schreibe Log-Eintrag <END CKPT>
- Flush-Log

UNDO/REDO-Recovery:

?

Aufgabe 5:
Non-blocking Checkpointing

UNDO/REDO-Recovery:

Lese das Log rückwärts:

a. Lese zuerst <END CKPT>:
   - UNDO (umgekehrt chronologisch) aller Transaktionen ohne <COMMIT Ti> bis zum ersten <START Ti> der T1, …,Tk des <START CKPT>
   - REDO (chronologisch) aller Transaktionen Ti mit <COMMIT Ti>, die im vorhergegangenen <START CKPT (T1, …, Tk)> stehen oder danach gestartet wurden; zurückgehen nur bis zum <START CKPT> da frühere Änderungen der T1, …, Tk bereits geschrieben worden sein müssen

b. Lese zuerst <START CKPT (T1, …, Tk)>:
   - Suche den vorherigen <END CKPT> und nutze Vorgehen aus a), da dieser Log-Eintrag nicht bei der Recovery hilft

Aufgabe 6: Logging

	T1	T2	T3
1			start
2			read(B, t)
3			t` = t+1
4	start		
5	read(A, u)		
6			write(B, t`)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v` = v+w	
12		write(B, v`)	
13		commit	
14	read(B, x)		
15	x` = x+1		
16	write(B, x`)		Start Checkpoint
17	u` = u+1		
18	write(A, x)		
19	commit		

Gegeben:

- Transaktionen T1, T2 und T3

Aufgaben:

- Schreibe das zugehörige UNDO-Log
- Schreibe das zugehörige REDO-Log
- Schreibe das zugehörige UNDO/REDO-Log

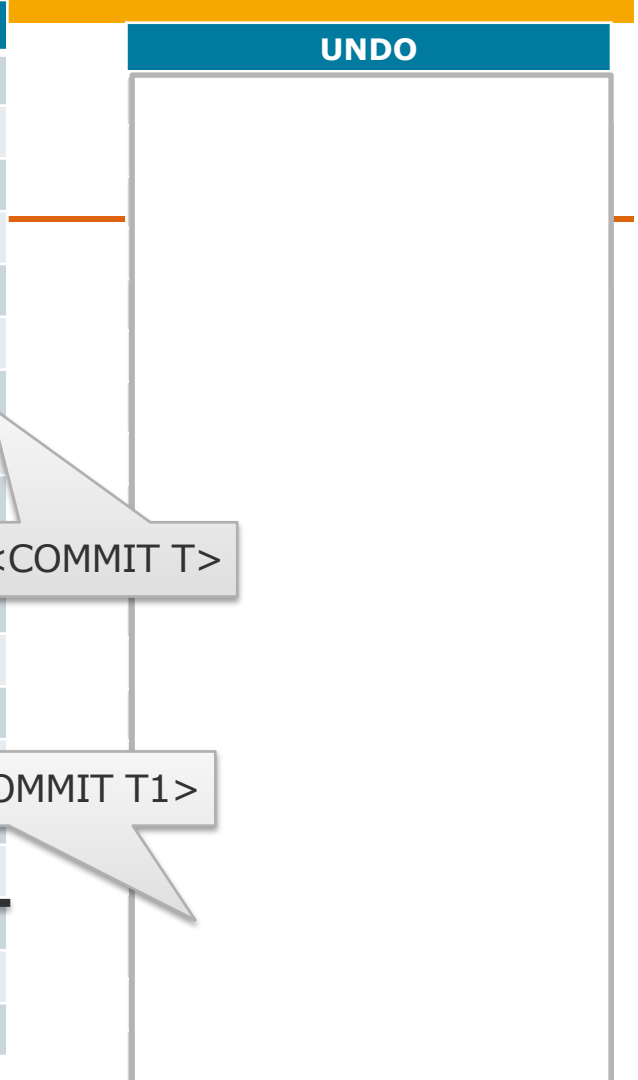
	T1	T2	T3
1			start
2			read(B, t)
3			t‘ = t+1
4	start		
5	read(A, u)		
6			write(B, t‘)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v‘ = v+w	
12		write(B, v‘)	
13		commit	
14	read(B, x)		
15	x‘ = x+1		
16	write(B, x‘)		
17	u‘ = u+1		
18	write(A, u‘)		
19	commit		

“commit“ bedeutet “output(X)“ aller Felder X der Transaktion

Annahme: Nicht-blockierendes Checkpointing

Start Checkpoint

	T1	T2	T3
1			start
2			read(B, t)
3			t` = t+1
4	start		
5	read(A, u)		
6			write(B, t`)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v` = v+w	
12		write(B, v`)	
13		commit	
14	read(B, x)		
15	x` = x+1		
16	write(B, x`)		
17	u` = u+1		
18	write(A, u`)		
19	commit		



	T1	T2	T3
1			start
2			read(B, t)
3			t` = t+1
4	start		
5	read(A, u)		
6			write(B, t`)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v` = v+w	
12		write(B, v`)	
13		commit	
14	read(B, x)		
15	x` = x+1		
16	write(B, x`)		
17	u` = u+1		
18	write(A, u`)		
19	commit		

T.commit vor <COMMIT T>

wartet auf <COMMIT T1>

UNDO
<START T3>
<START T1>
<T3,B,t>
<COMMIT T3>
<START T2>
<T2,B,v>
<COMMIT T2>
<T1,B,x>
<START CKPT (T1)> <T1,A,u>
<COMMIT T1><END CKPT>

	T1	T2	T3
1			start
2			read(B, t)
3			t` = t+1
4	start		
5	read(A, u)		
6			write(B, t`)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v` = v+w	
12		write(B, v`)	
13		commit	
14	read(B, x)		
15	x` = x+1		
16	write(B, x`)		
17	u` = u+1		
18	write(A, u`)		
19	commit		

REDO

<COMMIT T> vor T.commit

Muss nicht warten

	T1	T2	T3
1			start
2			read(B, t)
3			t‘ = t+1
4	start		
5	read(A, u)		
6			write(B, t‘)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v‘ = v+w	
12		write(B, v‘)	
13		commit	
14	read(B, x)		
15	x‘ = x+1		
16	write(B, x‘)		
17	u‘ = u+1		
18	write(A, u‘)		
19	commit		

<COMMIT T> vor T.commit

Muss nicht warten

REDO
<START T3>
<START T1>
<T3,B,t’>
<COMMIT T3>
<START T2>
<T2,B,v’>
<COMMIT T2>
<T1,B,x’>
<START CKPT (T1)>
<END CKPT> <T1,A,u‘>
<COMMIT T1>

	T1	T2	T3
1			start
2			read(B, t)
3			t` = t+1
4	start		
5	read(A, u)		
6			write(B, t`)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v` = v+w	
12		write(B, v`)	
13		commit	
14	read(B, x)		
15	x` = x+1		
16	write(B, x`)		
17	u` = u+1		
18	write(A, u`)		
19	commit		

UNDO/REDO

<COMMIT T> vor/nach T.commit

wartet bis Daten aus Puffer auf Disk sind

	T1	T2	T3
1			start
2			read(B, t)
3			t` = t+1
4	start		
5	read(A, u)		
6			write(B, t`)
7			commit
8		start	
9		read(B, v)	
10		read(A, w)	
11		v` = v+w	
12		write(B, v`)	
13		commit	
14	read(B, x)		
15	x` = x+1		
16	write(B, x`)		
17	u` = u+1		
18	write(A, u`)		
19	commit		

UNDO/REDO
<START T3>
<START T1>
<T3,B,t,t`>
<COMMIT T3>
<START T2>
<T2,B,v,v'>
<COMMIT T2>
<T1,B,x,x'>
<START CKPT (T1)>
<T1,A,u, u`>
<COMMIT T1>
<END CKPT>

<COMMIT T> vor/nach T.commit

wartet bis Daten aus Puffer auf Disk sind

Quiz: Richtig oder Falsch?
Logging Techniken

Hasso Plattner Institut

Beim UNDO-Logging werden Daten-I/O-Operationen häufiger ausgeführt als beim REDO-Logging.

Wahr Da die Daten sofort am Ende einer Transaktion geschrieben werden müssen um diese abschließen zu können, sind die I/O-Operationen beim UNDO-Logging häufiger.

Beim UNDO-Logging müssen alle veränderten Blocks im Puffer verbleiben bis Commit- und Change-Logs auf Disk geschrieben sind.

Falsch Dies gilt für das REDO-Logging! Beim UNDO-Logging werden die Daten-blocks vor dem Commit-Log auf die Festplatte geschrieben.

Ein REDO-Log lässt sich schneller wiederherstellen als ein UNDO-Log.

Falsch Da es in einem Log in der Regel wesentlich mehr abgeschlossene als offene Transaktionen gibt, dauert ein REDO aller abgeschlossenen Transaktionen länger als ein UNDO aller offenen Transaktionen.

Beim Recovery mit nicht-blockierendem Checkpointing muss das Log-File maximal bis zum ersten <START CKPT …> rückwärts durchlaufen werden.

21.01.19 / 23.01.19
Tobias Bleifuß

Falsch
UNDO: Falls <START CKPT …> vor <END CKPT> gelesen wird, müssen wir weiter rückwärts lesen.
REDO: Wir gehen in jedem Fall über das <START CKPT …> hinaus.

Aufgabenblatt 6

Aufgabe 1: UNDO-Log oder REDO-Log?

Betrachte das folgende Logfile

```
<START T1>
<T1, C, 35>
<T1, D, 450>
<START T2>
<T2, B, 40>
<COMMIT T1>
<START CKPT (T2)>
<END CKPT>
<T2, D, 18>
<START T3>
<T3, C, 18>
<T3, E, 18>
<T2, A, 13>
<COMMIT T3>
<COMMIT T2>
```

und die folgende auf der Festplatte gespeicherte Datenbank:

Element	Wert
A	13
B	40
C	35
D	4
E	18

a) Kann das gegebene Logfile ein UNDO-Log für die gegebene Datenbank sein? Warum? **2 P**

b) Kann das gegebene Logfile ein REDO-Log für die gegebene Datenbank sein? Warum? **2 P**

Aufgabenblatt 6

Aufgabe 2: Undo-Logging

Gegeben sei die folgende Sequenz von Undo-Log-Einträgen, die von vier Transaktionen S, T, U und V erzeugt wurden:

```
<START S>
<S, A, 60>
<COMMIT S>
<START T>
<T, A, 10>
<START U>
<U, B, 20>
<T, C, 30>
<START V>
<U, D, 40>
<V, F, 70>
<COMMIT U>
<T, E, 50>
<COMMIT T>
<V, B, 80>
<COMMIT V>
```

Aufgabenblatt 6

a) Beschreibe und begründe die vom Recovery-Manager auszuführenden Aktionen, wenn sich ein Fehler ereignet hat und der letzte auf der Festplatte geschriebene Log-Eintrag

1) `<START U>`
2) `<COMMIT U>`

ist. Gib zusätzlich für jeden der beiden Fälle an, welche von den Transaktionen geschriebenen Werte bereits auf die Festplatte geschrieben sein *müssen*. Welche Werte *könnten* bereits auf die Festplatte geschrieben worden sein? **3+3 P**

b) Füge den Start eines nicht-blockierenden Checkpointing direkt nach den folgenden Einträgen ein:

1) `<U, B, 20>`
2) `<T, E, 50>`

und beantworte für beide Fälle die folgenden Fragen:

- Wie lautet der komplette Log-Eintrag für den Start des Checkpointing?
- Wo wird der

  `<END CKPT>`

  Log-Eintrag eingefügt?
- Für alle möglichen Fehlerfälle (nach dem Start des Checkpointing): Bis zu welchem Eintrag muss das Logfile beim Recovery gelesen werden?

  Hinweis: Was ist das wesentliche Unterscheidungsmerkmal für die möglichen Fehlerfälle? **4+4 P**

Aufgabe 3: Undo/Redo-Logging

Gegeben sei die folgende Folge von Undo/Redo-Log-Einträgen, die von vier Transaktionen S, T, U und V erzeugt wurden:

```
<START S>
<S, A, 60, 61>
<COMMIT S>
<START T>
<T, A, 61, 62>
<START U>
<U, B, 20, 21>
<T, C, 30, 31>
<START V>
<U, D, 40, 41>
<V, F, 70, 71>
<COMMIT U>
<T, E, 50, 51>
<COMMIT T>
<V, B, 21, 22>
<COMMIT V>
```

Aufgabenblatt 6

a) Beschreibe und begründe die vom Recovery-Manager auszuführenden Aktionen, wenn sich ein Fehler ereignet hat und der letzte auf der Festplatte geschriebene Log-Eintrag

1) `<START U>`

2) `<COMMIT U>`

ist. Gib zusätzlich für jeden der beiden Fälle an, welche von den Transaktionen geschriebenen Werte bereits auf die Festplatte geschrieben sein *müssen*. Welche Werte *könnten* bereits auf die Festplatte geschrieben worden sein? **3+3 P**

b) Füge den Start eines Checkpointing direkt nach den folgenden Einträgen ein:

1) `<U, B, 20, 21>`

2) `<T, E, 50, 51>`

und beantworte für beide Fälle die folgenden Fragen:

- Wie lautet der komplette Log-Eintrag für den Start des Checkpointing?
- Wo kann der

  `<END CKPT>`

  Log-Eintrag eingefügt werden?
- Für alle möglichen Fehlerfälle (nach dem Start des Checkpointing): Bis zu welchem Eintrag muss das Logfile beim Recovery gelesen werden?

  Hinweis: Was ist das wesentliche Unterscheidungsmerkmal für die möglichen Fehlerfälle? **4+4 P**