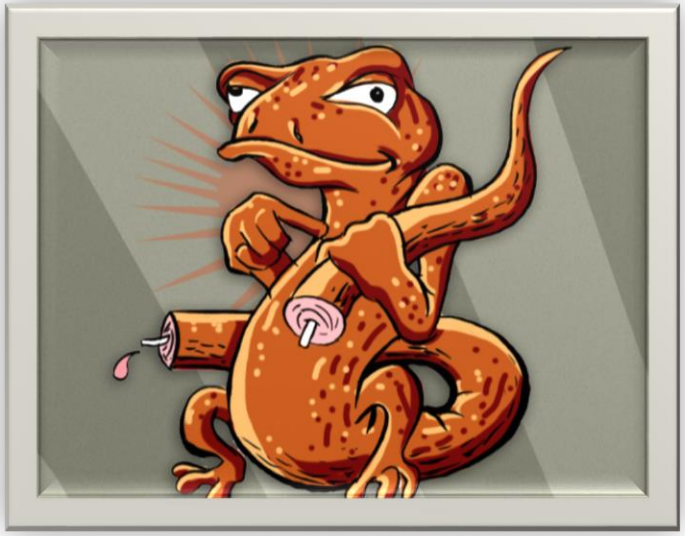
Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Übung Datenbanksysteme II

Recovery



Maximilian Jenders

Folien basierend auf
Thorsten Papenbrock

Wdh:
Warum Recovery?

Widerstandsfähigkeit (*Resilience*)

- Integrität der Datenbank trotz Fehler wiederherstellen
  - Systemfehler => Logging
  - Medienfehler => Archivierung
- Fehlerarten (nun etwas feiner)
  - Fehlerhaften Dateneingabe
  - Medienfehler
  - Katastrophe
  - Systemfehler

Systemfehler

- Transaktionen
  - Haben während des Ablaufs einen Zustand
    - Werte für Variablen
    - Aktuelles Code-Stück
  - Zustand im Hauptspeicher
  - Zustand geht bei Systemfehler verloren
- Stromausfall
  - Zustand gelöscht
- Softwarefehler
  - Zustand überschrieben

- Abhilfe
  - Logging

Wdh: Transaktionen

Wdh: Die Transaktion

- Eine Transaktion ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten (eventuell veränderten) Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss.

- Wdh.: Wofür steht ACID?

- Zwei Aspekte:
  - Semantische Integrität: Korrekter (konsistenter) DB-Zustand nach Ende der Transaktion
  - Ablaufintegrität: Fehler durch „gleichzeitigen" Zugriff mehrerer Benutzer auf dieselben Daten vermeiden
  - Widerstandfähigkeit: Häufiges Schreiben auf Disk
  - Effizienz: Seltenes Schreiben auf Disk

Operationen

Operationen

- Bewegen von Elementen zwischen Adressräumen
- `INPUT(X)`
  - Kopiert `X` von Disk in Puffer
  - Ausgeführt von Puffermanager
- `READ(X, t)`
  - Kopiert `X` von Puffer in Transaktionsadressraum
  - `t` ist lokale Variable
  - Ausgeführt von Transaktion
- `WRITE(X, t)`
  - Kopiert Wert von `t` auf Element `X` im Puffer
  - Ausgeführt von Transaktion
- `OUTPUT(X)`
  - Kopiert Block mit `X` von Puffer auf Disk
  - Ausgeführt von Puffermanager
- Annahme: Ein Element passt immer in einen Block
  - Falls nicht zutreffend: Aufteilen von Elementen in einzelne Blöcke
  - Sorgfältig sein: Immer alle Blöcke schreiben

Logging

- Ziel: Atomizität der Transaktionen
- Log = Folge von „*log records*“ (Log-Datensätze)
- Aktionen verschiedener Transaktionen können sich überlappen.
  - Deshalb: Nicht erst am Ende der Transaktion die gesammelten Aktionen loggen.
- Systemfehler: Log zur Rekonstruktion eines konsistenten Zustandes verwenden
- Medienfehler: Log und Archiv verwenden
- Allgemein
  - Einige Transaktionen müssen wiederholt werden (redo).
  - Einige Transaktionen müssen rückgängig gemacht werden (undo).
- Log-Manager: *Flush-log* Befehl weist Puffermanager an, alle Log-Blöcke auf Disk zu schreiben.
- Transaktionsmanager: OUTPUT Befehl weist Puffermanager an, Element auf Disk zu schreiben.

Wdh:
Undo-Logging

Regeln des Undo-Logging

- U1: Falls Transaktion T Element X verändert, muss <T, X, v> auf Disk geschrieben sein BEVOR neuer Wert von X auf Disk geschrieben wird.
  - Puffermanager muss aufpassen.
- U2: Falls eine Transaktion committed: <COMMIT T> darf erst ins Log geschrieben werden NACHDEM alle veränderten Elemente auf Disk geschrieben wurden.
  - Puffermanager muss aufpassen.

Komplikation: Systemfehler während Recovery

- Was tun?

- Bei Systemfehler: Einfach nochmal von vorn anfangen
- Recovery-Schritte sind *idempotent*.
  - Wiederholte Anwendung entspricht einfacher Anwendung.
  - $f(f(x)) = f(x)$

Wdh: Checkpointing

Checkpointing

- Problem: Recovery zwingt, das gesamte Log zu lesen
  - Idee 1: Sowie man ein COMMIT sieht, abbrechen
    - Aber: Wegen paralleler Transaktionen können einige TAs dennoch uncommitted sein.
  - Idee 2: Periodisch *Checkpoint* setzen
    1. Vorübergehend neue Transaktionen ablehnen
    2. Warten bis alle laufenden Transaktion committed oder aborted sind und entsprechender Log-Datensatz geschrieben wurde
    3. Flush-log
    4. Log-Datensatz `<CKPT>` schreiben
    5. Flush-log
    6. Neue Transaktionen wieder annehmen
  - Änderungen aller vorigen Transaktionen sind auf Disk geschrieben.
  - Recovery muss nur bis `<CKPT>` Log lesen

Wdh: Checkpointing

Nicht-blockierendes Checkpointing

- Problem: Datenbank wird während des Checkpointing blockiert.
  - Keine neuen Transaktionen werden angenommen.
  - Laufende Transaktionen könnten noch viel Zeit benötigen.
- Idee: Checkpoint nur für bestimmte Transaktionen
  1. Schreibe Log-Datensatz `<START CKPT (T1, …, Tk)>`
     - Alle aktiven Transaktionen
  2. Flush-log
  3. Warte bis T1, …, Tk committed oder aborted sind
     - Erlaube aber neue Transaktionen!
  4. Schreibe Log-Datensatz `<END CKPT>`
  5. Flush-log

- Lesend von hinten erst <END CKPT> oder erst <START CKPT …>?

- Zuerst <END CKPT>
  - Recovery nur bis zum nächsten <START CKPT>
- Zuerst <START CKPT T1,…, Tk>
  - D.h. Systemfehler während Checkpointing
  - Zu diesem Zeitpunkt sind T1, …, Tk die einzigen aktiven Transaktionen.
  - Recovery weiter rückwärts…
    - … nur bis zum Start der zeitlich frühesten, non-committed Transaktionen aus T1, …, Tk

Wdh: Redo-Logging

- Log-Datensatz <T, X, v>
  - Transaktion T hat neuen Wert V für Datenbankelement X geschrieben.

- Redo Regel (Auch *write-ahead logging rule*)
  - R1: Bevor ein Datenbankelement X auf Disk verändert werden kann, müssen alle zugehörigen Log-Datensätze auf Disk geschrieben sein.
    - Zugehörig: Update-Datensatz und COMMIT
      - Undo macht unvollständige Transaktionen rückgängig und ignoriert vollständige Transaktionen.
      - Redo ignoriert unvollständige Transaktionen und wiederholt vollständige Transaktionen.

- COMMIT im Log
  - Undo: Erst nachdem alle Werte auf Disk sind
  - Redo: BEVOR irgendein Wert auf Disk geschrieben wird
    - Falls kein COMMIT im Log, sind Elemente auf Disk unberührt
      - Müssen also nicht wiederhergestellt werden
    - => Unvollständige Transaktionen können ignoriert werden.
  - Committed Transaktionen sind Problem
    - Unsicher, welche Änderungen auf Disk gelangt sind
    - Aber: Log-Datensätze haben alle Informationen

Redo-Logging

Redo-Logging – Checkpointing

- Neues Problem im Gegensatz zu Undo-Logging:
  - Beschränkung auf aktive Transaktionen genügt nicht:
    - COMMIT im Log obwohl Ergebnis noch nicht auf Disk geschrieben wurde.
- Idee: Während des Checkpointing alle Datenbankelemente auf Disk schreiben, die von committed aber noch nicht auf Disk geschriebenen Transaktionen verändert wurden.
  - Puffermanager muss *dirty* Puffer kennen.
  - Log-Manager muss wissen, welche Transaktion welchen Puffer verändert hat.
- Vorteil: Man muss nicht auf die Beendigung aktiver (also uncommitted) Transaktionen warten,
  - denn die dürfen zurzeit eh nichts schreiben.

Wdh:

Undo/Redo Logging

- Nachteil Undo: Daten müssen sofort nach Ende der Transaktion geschrieben werden => Mehr I/O
- Nachteil Redo: Alle veränderten Blocks müssen im Puffer verbleiben bis COMMIT und Log-Datensätze auf Disk sind => Hoher Speicherbedarf
- Undo/Redo Logging ist flexibler
  - Aber mehr Informationen in Log-Datensätzen nötig
  - Log-Datensatz: <T, X, v, w>
    - Alter Wert (v) und neuer Wert (w)
  - Regel UR1:
    - Update-Log-Datensatz <T, X, v, w> muss auf Disk geschrieben sein BEVOR das von T veränderte X auf Disk geschrieben wird.
  - Diese Bedingung wird von Undo und Redo gemäß voriger Regeln ebenfalls verlangt.
    - Reminder U1: Falls Transaktion T Element X verändert, muss <T, X, v> auf Disk geschrieben sein BEVOR neuer Wert von X auf Disk geschrieben wird.
    - Reminder R1: Bevor ein Datenbankelement X auf Disk verändert werden kann, müssen alle zugehörigen Log-Datensätze auf Disk geschrieben sein.

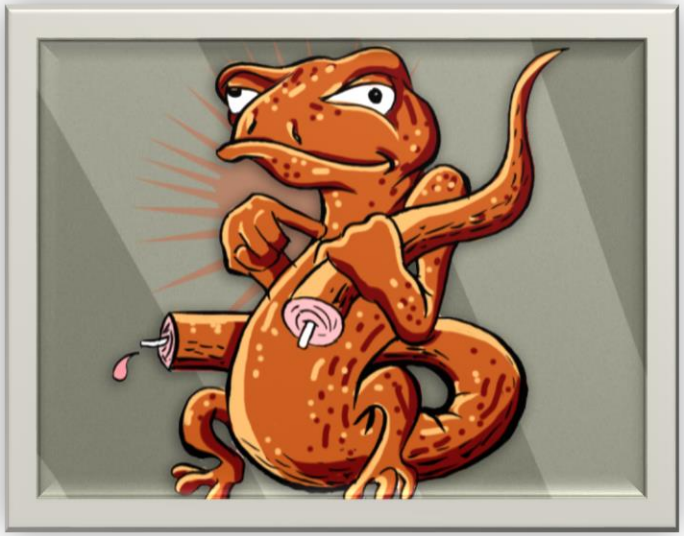
<COMMIT T> darf VOR oder NACH Änderungen der Elemente auf Disk geschrieben werden.

Undo/Redo Logging

Undo/Redo Recovery – Checkpointing

- Einfacheres Vorgehen
  - Schreibe <START CKPT (T1, …, Tk)> in Log
    - Ti sind alle aktiven Transaktionen
  - Flush-log
  - Schreibe auf Disk alle zum Beginn des Checkpoints *dirty* Puffer
    - Also solche, die mindestens ein verändertes Element enthalten
    - Anmerkung: Bei Redo nur die Puffer mit committed Transaktionen
  - Schreibe <END CKPT> in Log
  - Flush-log

Jetzt
Übung 6



- HA 5 Korrektur verzögert sich noch leicht, kommt bald!
- HA 6 wie besprochen: Abgabe entweder morgen oder 05.02.

Aufgabe 1:

Schreib-Reihenfolgen

- Relevante Schreib-Operationen auf Festplatte:
  a. Schreiben der *Daten*
  b. Schreiben der *Change*-Logs
  c. Schreiben des *Commit*-Logs

- In welcher Reihenfolge werden die Operationen ausgeführt bei:

UNDO	REDO	UNDO/REDO
1. Change 2. Daten 3. Commit	1. Change 2. Commit 3. Daten	1. Change 2. Daten / Commit

Aufgabe 2: Recovery-Strategie

- Wie erfolgt die Wiederherstellung eines konsistenten Zustands bei:

UNDO	REDO	UNDO/REDO
1. Change 2. Daten 3. Commit	1. Change 2. Commit 3. Daten	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	REDO	UNDO/REDO
1. Change 2. Daten 3. Commit	1. Change 2. Commit 3. Daten	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	REDO	UNDO/REDO
1. Change 2. Daten 3. Commit	1. Change 2. Commit 3. Daten	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:
  1. Schreibe Log-Eintrag <START CKPT (T1, …, Tk)>
     - T1, …, Tk sind alle aktiven Transaktionen
  2. Flush-Log
  3. Warte auf <COMMIT Ti> bzw. <ABORT Ti> aller T1, …, Tk
     - Erlaube dabei neue Transaktionen!
  4. Schreibe Log-Eintrag <END CKPT>
  5. Flush-Log

- UNDO-Recovery: **?**

Non-blocking Checkpointing

- UNDO-Recovery:

Lese den 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:
  1. Schreibe Log-Eintrag <START CKPT (T1, …, Tk)>
     - T1, …, Tk sind alle aktiven Transaktionen
     - Aktiv: Commit noch nicht im Log!
  2. Flush-Log
  3. 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!
  4. Schreibe Log-Eintrag <END CKPT>
  5. Flush-Log

?

- REDO-Recovery:

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:
  1. Schreibe Log-Eintrag <START CKPT (T1, …, Tk)>
     - T1, …, Tk sind alle aktiven Transaktionen
  2. Flush-Log
  3. 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!
  4. Schreibe Log-Eintrag <END CKPT>
  5. Flush-Log
- UNDO/REDO-Recovery: **?**

Non-blocking Checkpointing

- UNDO/REDO-Recovery:

Lese den 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:
  1. Schreibe das zugehörige UNDO-Log
  2. Schreibe das zugehörige REDO-Log
  3. 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		

UNDO

T.commit vor <COMMIT T>

wartet auf <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
<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>

T.commit vor <COMMIT T>

wartet auf <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		

REDO

<COMMIT T> vor T.commit

wartet auf T2.commit
und T3.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		

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>

<COMMIT T> vor T.commit

wartet auf T2.commit
und T3.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		

UNDO/REDO

<COMMIT T> vor/nach T.commit

wartet auf T1.commit,
T2.commit und T3.commit

	T1	T2	T3	UNDO/REDO
1			start	
2			read(B, t)	<START T3>
3			t‘ = t+1	
4	start			
5	read(A, u)			<START T1>
6			write(B, t‘)	<T3,B,t,t‘>
7			commit	<COMMIT T3>
8		start		
9		read(B, v)		<START T2>
10		read(A, w)		
11		v‘ = v+w		
12		write(B, v‘)		<T2,B,v,v’>
13		commit		<COMMIT T2>
14	read(B, x)			
15	x‘ = x+1			
16	write(B, x‘)			<T1,B,x,x’>
17	u‘ = u+1			<START CKPT (T1)>
18	write(A, u‘)			<T1,A,u, u‘>
19	commit			<COMMIT T1>
				<END CKPT>

<COMMIT T> vor/nach T.commit

wartet auf T1.commit, T2.commit und T3.commit

Logging Techniken

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.

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.

Abgabefrist für schnelle Bearbeitung von Übung 6 morgen!

Danach Musterlösung per Mailingliste

Übung 7 zu WDM findet nicht statt, Folien + HA + Musterlösung werden online gestellt

Weiter Fragen:

Mailingliste!

Klausurvorbereitung am 01.02. !

Evaluierung nicht vergessen!