Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

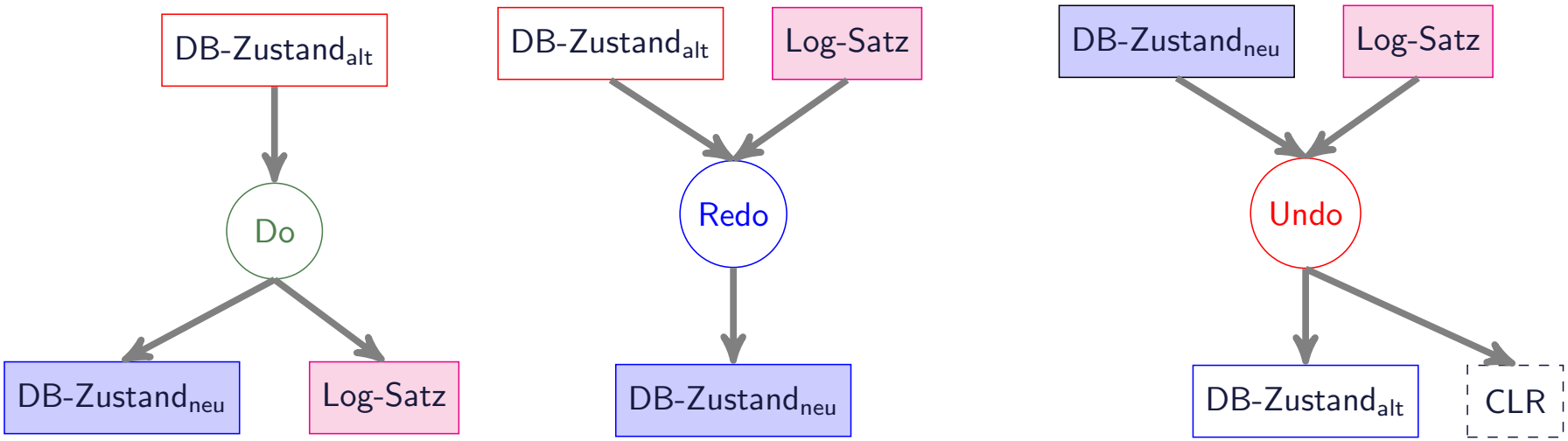
smichel@cs.uni-kl.de

Vorsorge für den Fehlerfall

Logging

- Sammlung redundanter Daten bei Änderungen im Normalbetrieb, als Voraussetzung für Recovery
- Einsatz im Fehlerfall (Undo-, Redo-Recovery)

Do-Redo-Undo-Prinzip



Crash-Recovery

Im folgenden wird Crash-Recovery betrachtet.

Idee/Ziel

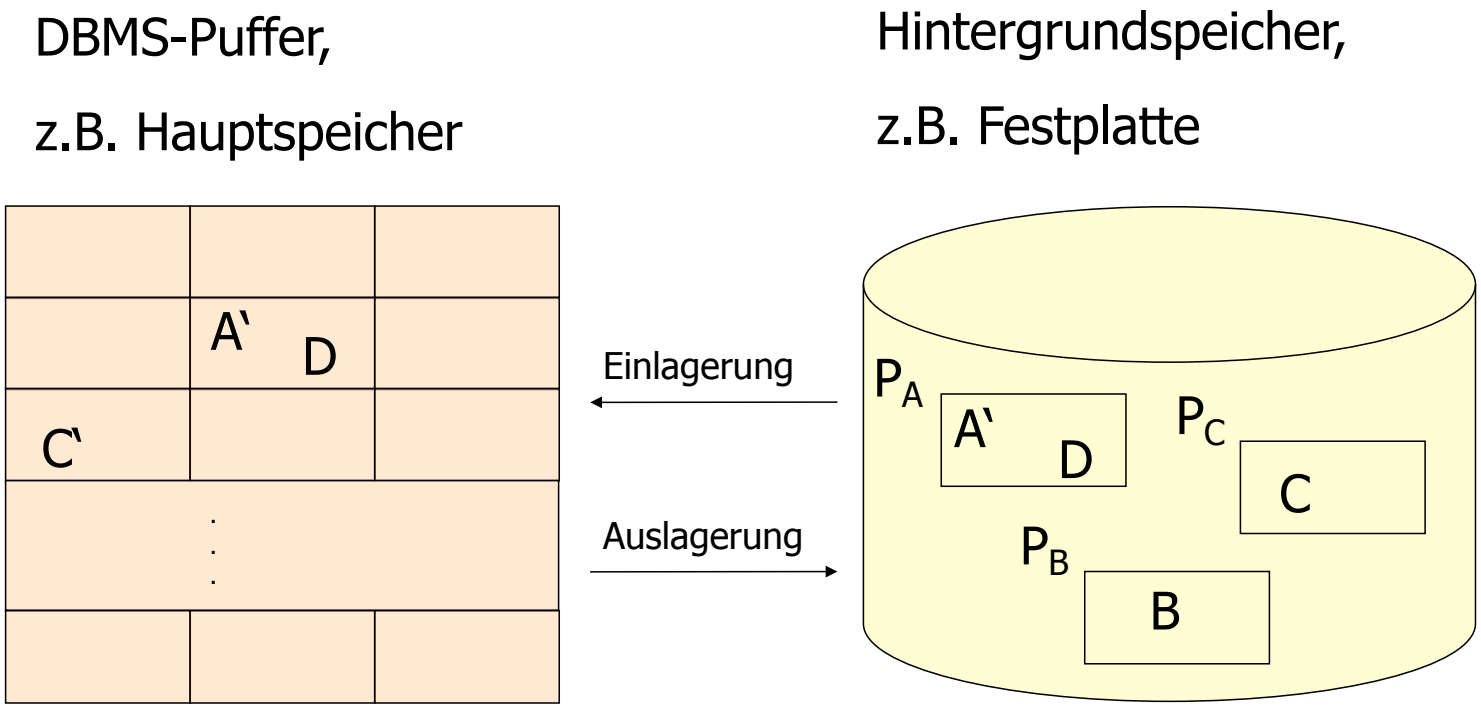
- Wiederanlauf (Restart) des DBS nach einem Systemfehler
- Ziel: Nach Wiederanlauf den jüngsten transaktionskonsistenzen DB-Zustand wiederherstellen, der zum Fehlerzeitpunkt gültig war.
- Dazu wird materialisierte Datenbasis und Log-Datei benutzt.
- Zusätzliche Anforderung: Idempotenz! D.h. bei mehrfacher Anwendung der Recovery muss dies stets zum selben Ergebnis führen.

Abhängigkeit von System-Konfiguration

- Methoden zur Crash-Recovery sind wesentlich durch die zugrunde liegende System-Konfiguration bestimmt (Satz oder Seitensperren, steal oder no steal, etc), wie auf folgenden Folien motiviert.

Zweistufige Speicherhierarchie

- Seiten P_i
- Datensätze $A, B, C, D, ..$
- **Werden später den Fall betrachten, dass Datensätze gesperrt werden, also versch. TA die gleiche Seite bearbeiten können.**



Hier zugrunde gelegte Systemkonfiguration

- **steal**
  "dreckige Seiten" können in die Datenbank (auf Platte) geschrieben werden. Wir brauchen also **Undo**!
- **¬force**
  geänderte Seiten sind **möglicherweise** noch nicht auf die Platte geschrieben. Wir brauchen also **Redo**!
- **update-in-place**
  Es gibt von jeder Seite nur eine Kopie auf der Platte
- **Kleine Sperrgranulate, kleiner als eine Seite**
  auf Satzebene. Also kann eine Seite gleichzeitig "dreckige" Daten (einer noch nicht abgeschlossenen TA) und "committed updates" enthalten.

WAL-Prinzip und Commit-Regel bei Log-basierter Recovery

Write-Ahead-log-Prinzip (WAL)

Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in die Log-Datei und das Log-Archiv ausgeschrieben werden.

Commit-Regel (Force-Log-at-Commit)

Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle "zu ihr gehördenden" Log-Einträge auf stabilen Storage ausgeschrieben werden.

C. Mohan et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.
`http://www.cs.berkeley.edu/~brewer/cs262/Aries.pdf`

Beispiel einer Log-Datei

Schritt	T_1	T_2	Log-Record [LSN, TA, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1,T_1, **BOT**, 0]
2.	r(A,a1)		
3.		**BOT**	[#2,T_2, **BOT**, 0]
4.		r(C,c2)	
5.	a1 := a1 - 50		
6.	w(A,a1)		[#3,T_1,PA,A-=50,A+=50,#1]
7.		c2 := c2 + 100	
8.		w(C,c2)	[#4,T_2,PC,C+=100,C-=100,#2]
9.	r(B,b1)		
10.	b1 := b1 + 50		
11.	w(B,b1)		[#5,T_1,PB,B+=50,B-=50,#3]
12.	**commit**		[#6,T_1,**commit**,#5]
13.		r(A,a2)	
14.		a2 := a2 - 100	
15.		w(A,a2)	[#7,T_2,PA,A-=100,A+=100,#4]
16.		**commit**	[#8,T_2,**commit**,#7]

PageLSN zur Erkennung ob Before oder AfterImage

LSN des Log-Eintrags wird mitkopiert (in Seite), wenn Seite auf Hintergrundspeicher propagiert wird. Daran kann man dann erkennen, ob für einen bestimmten Log-Eintrag das Before-Image oder After-Image in der Seite steht:

- Wenn die LSN der Seite (genannt PageLSN) einen kleineren Wert als die LSN des Log-Eintrags enthält, handelt es sich um das Before-Image.
- Ist die LSN der Seite größer oder gleich der LSN des Log-Eintrags, dann war schon das After-Image bzgl. der protokollierten Änderungsoperation auf den Hintergrundspeicher propagiert worden.

Also, für Log-Eintrag L und Seite B:

if LSN(L) $>$ PageLSN(B) **then do**
    REDO (Änderung aus L)
    PageLSN(B) := LSN(L)
end

Was wissen wir nach Absturz und was ist zu tun?

Verfügbare Informationen

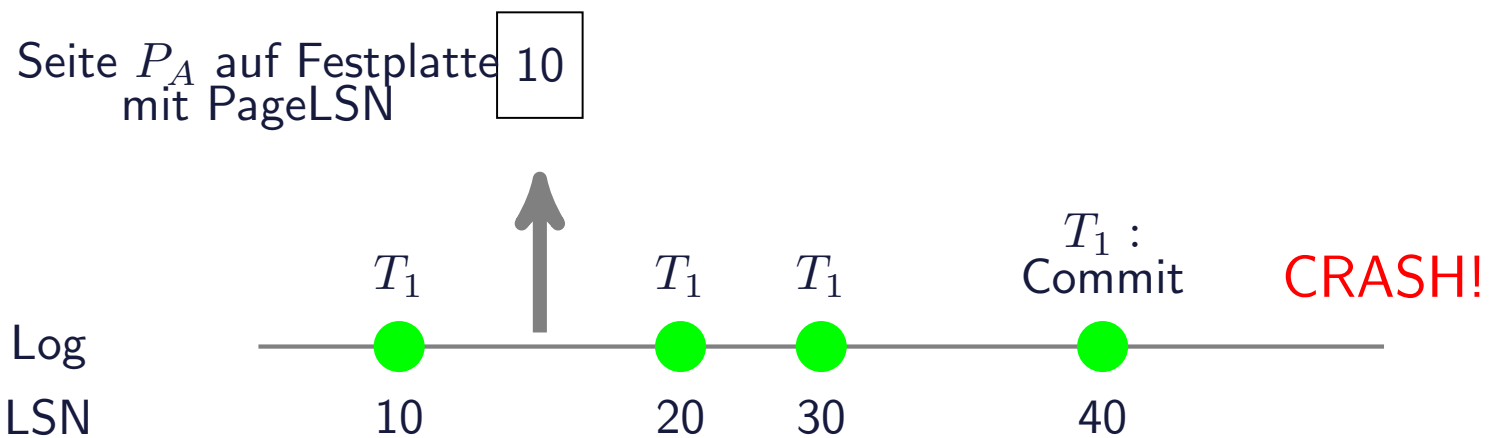
- Wir sehen das Log. Wir kennen die WAL-Regel und die Commit-Regel, also es kann keine Änderung eine Seite auf der Festplatte geändert haben ohne dass wir die Änderungsoperation im Log sehen!
- Wir haben für die Seiten auf der Festplatte die LSN (PageLSN) der Aktion, die zuletzt die Seite geändert hat.

Was ist zu tun/analysieren?

- Welche der protokollierten Änderungen wurden bereits ausgeführt?
- Welche der Änderungen müssen ausgeführt werden, weil wir zwar den Log-Eintrag sehen, aber die Änderung vor Absturz noch nicht auf Festplatte geschrieben wurde?
- Welche der Änderungen müssen rückgängig gemacht werden?

Beispiel

- Betrachten wir eine einzelne Transaktion T_1, welche einen Datensatz in einer Seite P_A bearbeitet.
- Die Punkte beschreiben Log-Einträge von Änderungsoperationen
- Nach dem Eintrag mit LSN 10 wird die Seite auf die Festplatte ausgeschrieben (genannt flush), z.B. weil sie verdrängt wurde.
- D.h. die PageLSN wird auf 10 gesetzt.
- Weitere Änderungsoperationen für LSN 20 und 30, aber kein Ausschreiben! Also Änderung nur im DB-Puffer.
- Wiederanlauf: Redo für Operationen mit LSN 20 und 30, aber Redo für LSN 10 nicht nötig.



Drei Phasen des Wiederanlaufs

1. Analyse (Bestimmung des Datenbankzustands)

- Die Log-Datei wird von Anfang bis zum Ende analysiert,
- Ermittlung der Winner-Menge von Transaktionen des Typs T1
- Ermittlung der Loser-Menge von Transaktionen der Art T2.

2. Redo (Vollständige Wiederholung der Historie)

- Alle protokollierten Änderungen werden in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht.
- Auch die Änderungen der Loser!

3. Undo (Entfernen der Loser-Änderungen)

- Die Änderungsoperationen der Loser-Transaktionen werden in umgekehrter Reihenfolge ihrer ursprünglichen Ausführung rückgängig gemacht.

Fehlertoleranz (Idempotenz) des Wiederanlaufs

Zu jeder ausgeführten Aktion a gilt:

- $undo(undo(...(undo(a))...)) = undo(a)$
- $redo(redo(...(redo(a))...)) = redo(a)$

Achtung: auch während der Recoveryphase kann das System abstürzen!

- Recovery funktioniert auch dann!

Fehlertoleranz (Idempotenz) des Wiederanlaufs (2)

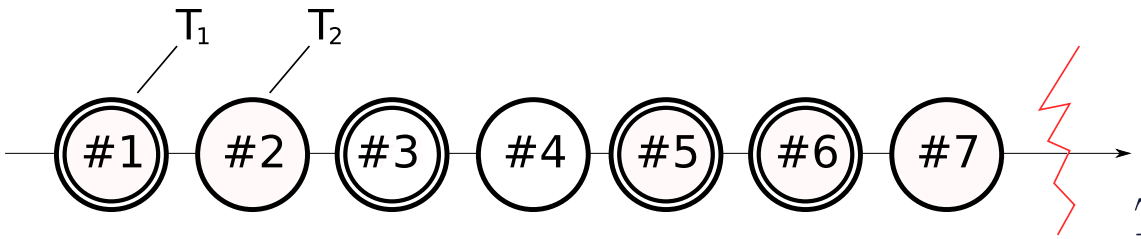
Der Redo Fall:

- LSN des Log-Records, für den ein Redo (**tatsächlich**) ausgeführt wird, wird in der Seite (PageLSN!) eingetragen
- Dadurch: nach Absturz nicht "versehentlich" nochmal ausgeführt

Der Undo Fall:

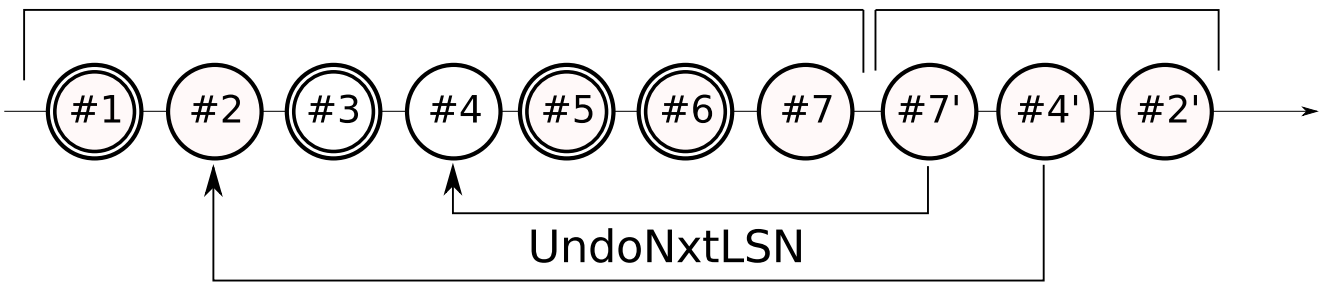
- Kompensations-Protokolleinträge (**CLR**, compensation log record)
- Für jede Undo-Operation wird ein CLR angelegt.
- Auch hier: LSN des CLR-Log-Eintrags wird als PageLSN übernommen.

Kompensationseinträge im Log



$T_1 =$ Gewinner, $T_2 =$ Verlierer

Wiederanlauf und Log



Kompensationseinträge (CLR: compensating log record) für rückgängig gemachte Änderungen.

- $\#7'$ ist CLR für $\#7$
- $\#4'$ ist CLR für $\#4$
- Achtung: Natürlich müssen alle LSN fortlaufend sein (hier nur zur Veranschaulichung z.B. $4'$ genannt)

Logeinträge nach abgeschlossenem Wiederanlauf

[#1, T_1, BOT, 0]
[#2, T_2, BOT, 0]
[#3,T_1, PA, A-=50, A+=50, #1]
[#4,T_2, PC, C+=100, C-=100, #2]
[#5,T_1, PB, B+=50, B-=50, #3]
[#6,T_1, commit, #5]
[#7,T_2, PA, A-=100, A+=100,#4]
<#7‘,T_2, PA, A+=100, #7,#4>
<#4‘,T_2, PC, C-=100, #7’,#2>
<#2‘,T_2,-,-,#4‘,0>

Logeinträge nach abgeschlossenem Wiederanlauf

CLRs sind durch spitze Klammern $< ... >$ gekennzeichnet. Und haben folgenden Aufbau:

- LSN
- ID der Transaktion
- betroffene Seite
- Redo-Information
- PrevLSN
- UndoNxtLSN (Verweis auf die nächste rückgängig zu machende Änderung)

Anmerkungen

- **Die Redo-Anweisung eines CLR entspricht der während der Undo-Phase des Wiederanlaufs ausgeführten Undo-Operation.**
- CLRs enthalten keine Undo-Information. Warum?

Idempotenz des Wiederanlaufs

[#1, T_1, BOT, 0]
[#2, T_2, BOT, 0]
[#3,T_1, PA, A-=50, A+=50, #1]
[#4,T_2, PC, C+=100, C-=100, #2]
[#5,T_1, PB, B+=50, B-=50, #3]
[#6,T_1, commit, #5]
[#7,T_2, PA, A-=100, A+=100,#4]
<#7‘,T_2, PA, A+=100, #7,#4>
<#4‘,T_2, PC, C-=100, #7’,#2>
<#2‘,T_2,-,-,#4‘,0>

- Was passiert bei Wiederanlauf?
- Log wie links gegeben.
- **Redo Phase** schaut ob alle “normalen” Änderungen durchgeführt sind und ebenso ob die Kompensationen ausgeführt wurden.
- **Undo Phase** sieht, dass UndoNxtLSN von $\#2'$ Null ist und merkt, dass T_2 vollständig rückgängig gemacht wurde.

Idempotenz des Wiederanlaufs (2)

Jetzt: Log Datei nur bis (einschließlich) $\#7'$

[#1, T_1, BOT, 0]
[#2, T_2, BOT, 0]
[#3,T_1, PA, A-=50, A+=50, #1]
[#4,T_2, PC, C+=100, C-=100, #2]
[#5,T_1, PB, B+=50, B-=50, #3]
[#6,T_1, commit, #5]
[#7,T_2, PA, A-=100, A+=100,#4]
<#7',T_2, PA, A+=100, #7,#4>

- Was passiert bei Wiederanlauf?
- **Redo Phase** schaut ob alle "normalen" Änderungen durchgeführt sind und ebenso ob die Kompensation $\#7'$ ausgeführt wurde.

- In **Undo Phase** wird offensichtlich $\#7'$ nicht rückgängig gemacht, da es ja ein CLR ist. Sondern es wird der UndoNxtLSN-Zeiger benutzt und gemerkt, dass die nächste Operation die rückgängig gemacht werden muss Operation $\#4$ ist. Die in LSN 4 stehende Undo-Operation wird ausgeführt, wobei der CLR $\#4'$ angelegt wird. In $\#4$ steht PevLSN-Veweis auf $\#2$, die als nächstes kompensiert wird und mit $\#2'$ protokolloert wird.

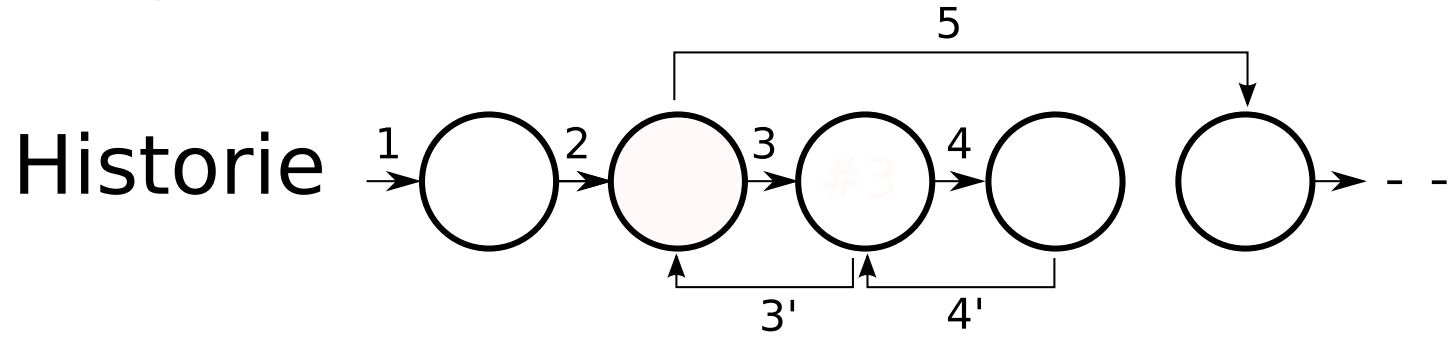
Lokales Zurücksetzen einer Transaktion

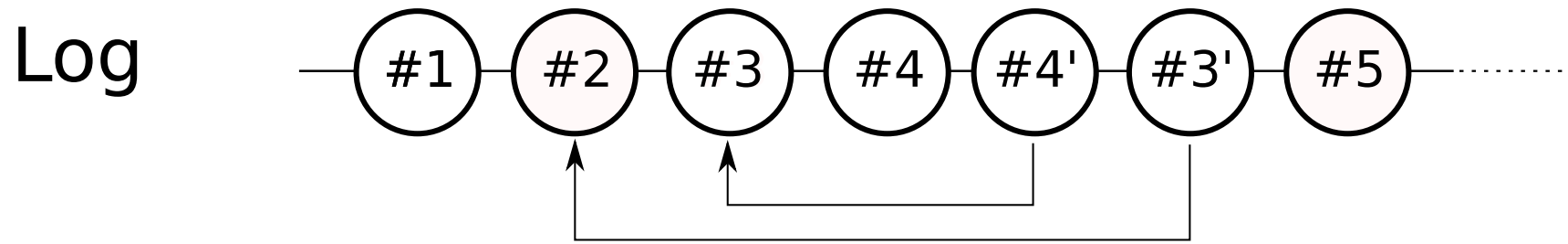
Isoliertes Zurücksetzen einer einzelnen Transaktion

- Analog zum Rücksetzen von Verlierer-Transaktionen nach Wiederanlauf
- Abarbeiten der zur Transaktion gehörenden Log-Einträge in chronologisch umgekehrter Reihenfolge.
- Dies kann nun aus dem Log-Puffer geschehen, da hier davon ausgegangen wird, dass der Hauptspeicher intakt ist.
- Mit den PrevLSN kann man sehr effizient zurücklaufen
- Und Undo-Operationen ausführen
- Aber: Vorher noch mit Compensation Log Record protokollieren!

Lokales Zurücksetzen einer Transaktion (2)

Z.B. partielles Zurücksetzen einer Transaktion





- Schritte 3 und 4 werden zurückgenommen
- notwendig für die Realisierung von **savepoints** einer TA

Lokales Zurücksetzen einer Transaktion (3)

- Zurücksetzen durch Einfügen von Operationen, die die Änderungen rückgängig machen.
- Also "kompensieren".

Vorgehen

- Transaktion "sagt": abort
- Log-Eintrag für abort Operation wird angelegt
- Kompensations-Operationen (Undo) der TA werden ausgeführt und (natürlich!) protokolliert, mit CLRs.
- Auch wird wieder die LSN des CLR-Eintrags als PageLSN übernommen!

Lokales Zurücksetzen einer Transaktion (4)

Absturz während Abort

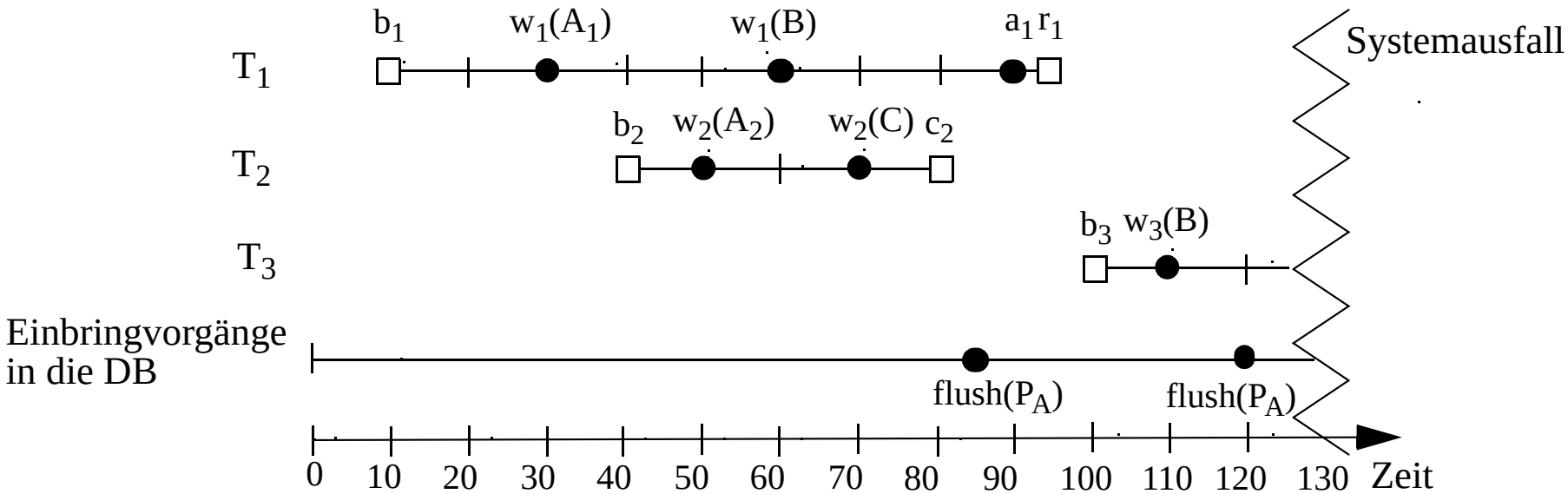
- Die CLRs werden im Falle eines Wiederanlaufs in der Redo-Phase überprüft ob bereits ausgeführt.
- In der Undo-Phase werden, wenn nötig, noch nicht protokollierte Kompensationen (wie bei allen Verlierer-TA) rückgängig gemacht.

Rollback Log-Eintrag

- Sozusagen als Commit (Festschreiben) einer Transaktion die Abort ausgeführt hat
- Mit Log Eintrag `rollback` (oder r)
- (Komplett) zurücksgesetzte Transaktion wird dann als Gewinner betrachtet, bei Wiederanlauf.
- Alternativ (imlizit): UndoNextLSN=Null?
- Literatur nicht immer eindeutig bzw. konkret diesbzgl.

Komplexeres Beispiel

- Transaktionen T_1, T_2 und T_3
- Datensatz A in Seite P_A, B in P_B und C in P_C
- Zwei Mal "flush" der Seite P_A
- Wie sieht Log-Puffer, Log-Datei, DB-Puffer und Seite auf Festplatte aus?



Komplexeres Beispiel → Übungsblatt 12: Ausfüllen

Zeit	Aktion	DB-Puffer	DB-Eintrag	Log-Puffer	Log-Datei
10	b_1				
30	$w_1(A_1)$				
40	b_2				
50	$w_2(A_2)$				
60	$w_1(B)$				
70	$w_2(C)$				
80	c_2				
85	flush(P_A)				
90	a_1				
91					
92					
95	r_1				
100	BOT_3				
110	$w_3(B)$				
120	flush(P_A)				

Absturzpunkt

Sicherungspunkte (=checkpoints)

Beobachtung

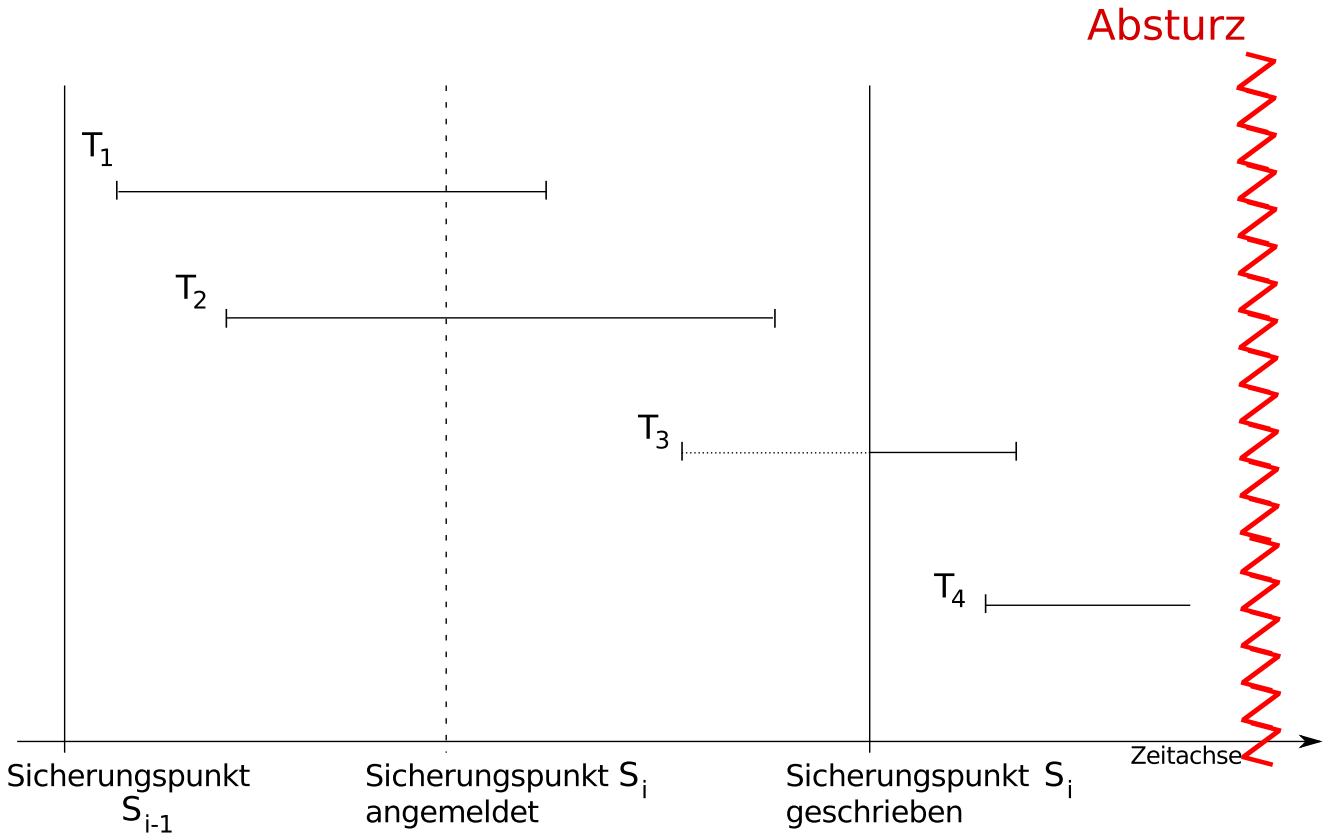
- Mit zunehmender Betriebszeit des Datenbanksystems wird Wiederanlauf immer langwieriger, da die Log-Datei immer umfangreicher wird.

Sicherungspunkte

- Idee: "Markiere" im Log Zeitpunkt, über die man beim Wiederanlauf nicht hinausgehen muss.
- Verschiedene Ansätze.
  - (globale) transaktionskonsistente Sicherungspunkte
  - aktionskonsistente Sicherungspunkte und
  - unscharfe (fuzzy) Sicherungspunkte
- Achtung: "cut-off" Punkt ist nicht unbedingt Zeitpunkt an dem Sicherungspunkt angelegt wird, kann auch älter sein; kleinste noch benötigte LSN wird angegeben.

Transaktionskonsistente Sicherungspunkte (=checkpoints)

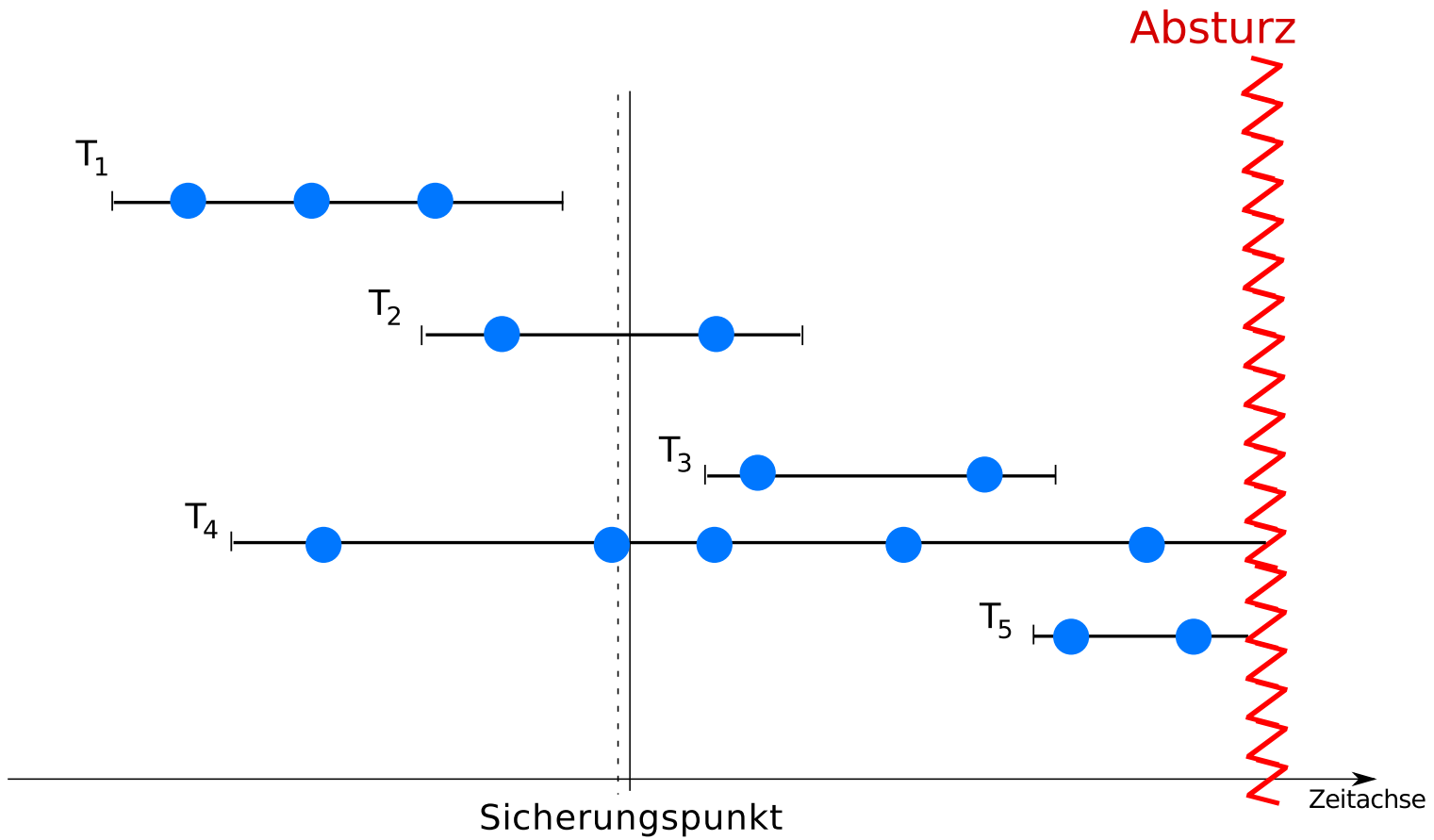
- Neu ankommende TA müssen warten. Laufende TA werden zu Ende ausgeführt.
- Dann schreiben aller modifizierten Seiten auf Hintergrundspeicher.



Achtung: checkpoint ≠ savepoint (Terminologie)

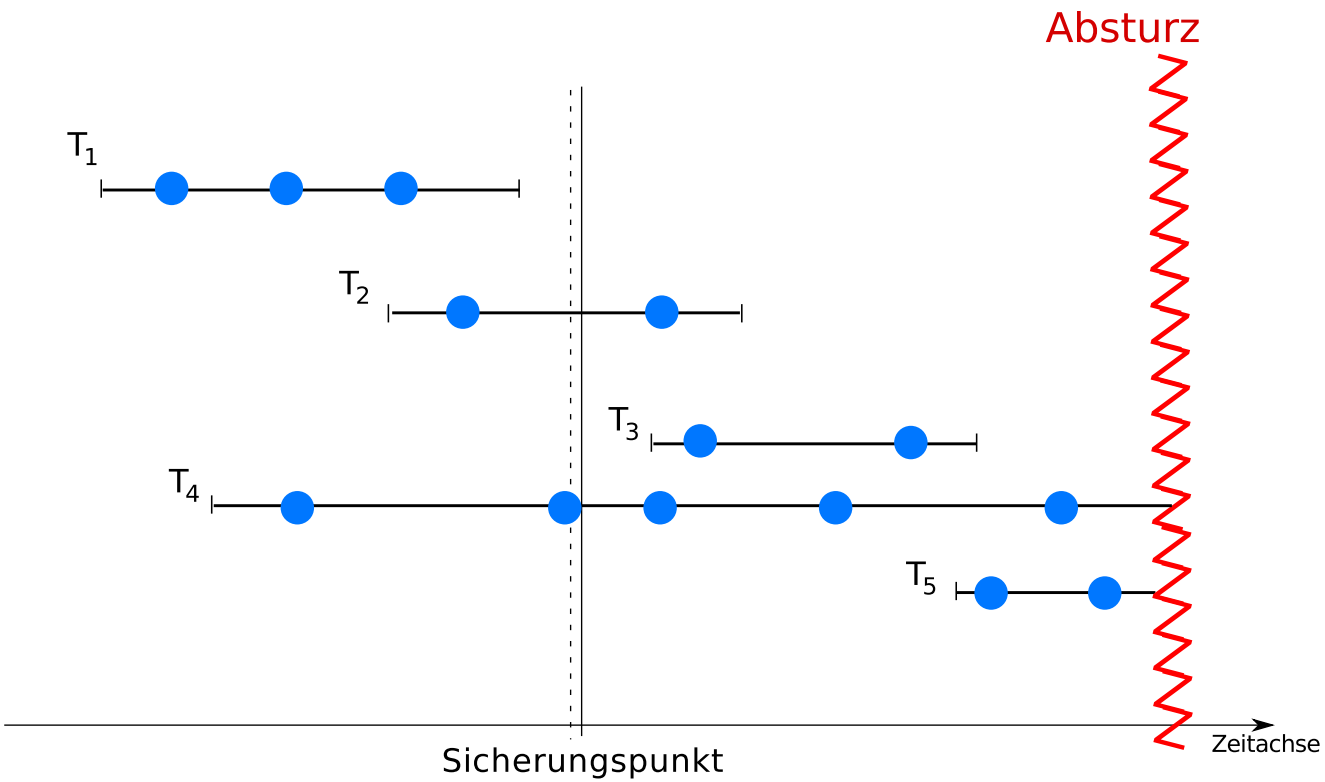
Aktionskonsistente Sicherungspunkte

- Transaktionsausführung relativ zu einem aktionskonsistenten Sicherungspunkt und einem Systemabsturz.
- Änderungsoperationen (blaue Punkte unten) werden noch ausgeführt
- Dann ausgeschrieben



Aktionskonsistente Sicherungspunkte (2)

- Man braucht keine Redo-Informationen, die älter sind als der Zeitpunkt des Schreibens auf Hintergrundspeicher
- Aber man braucht u. U. Undo-Informationen
- Also: Kleinste LSN aller zum Zeitpunkt noch aktiven LSN speichern (=MinLSN) + Liste aller noch aktiven TA



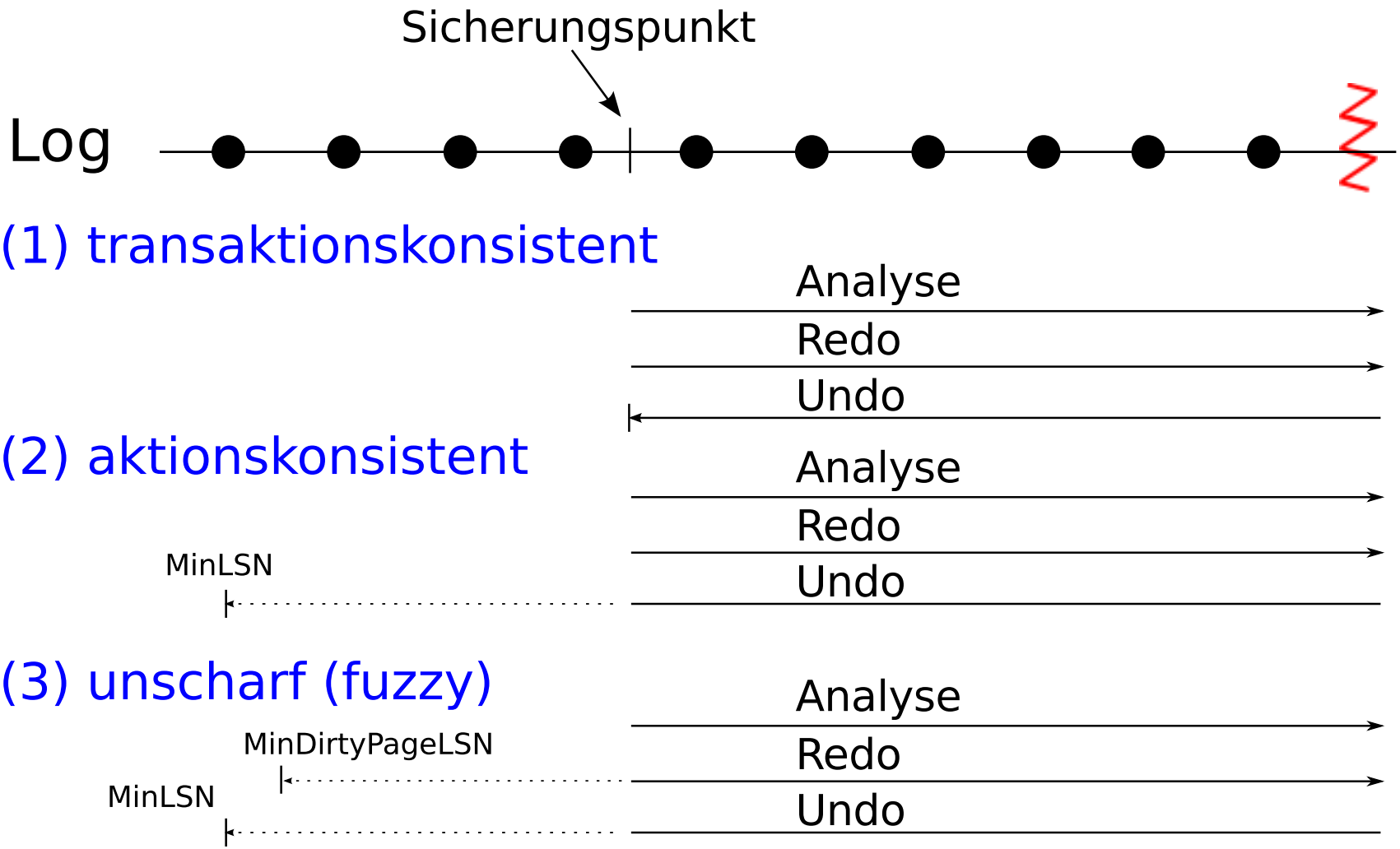
Unscharfe (fuzzy) Sicherungspunkte

- Idee: modifizierte Seiten werden **nicht** ausgeschrieben
- Sondern nur deren Kennung (PageID) wird notiert (in Log-Datei).

Terminologie:

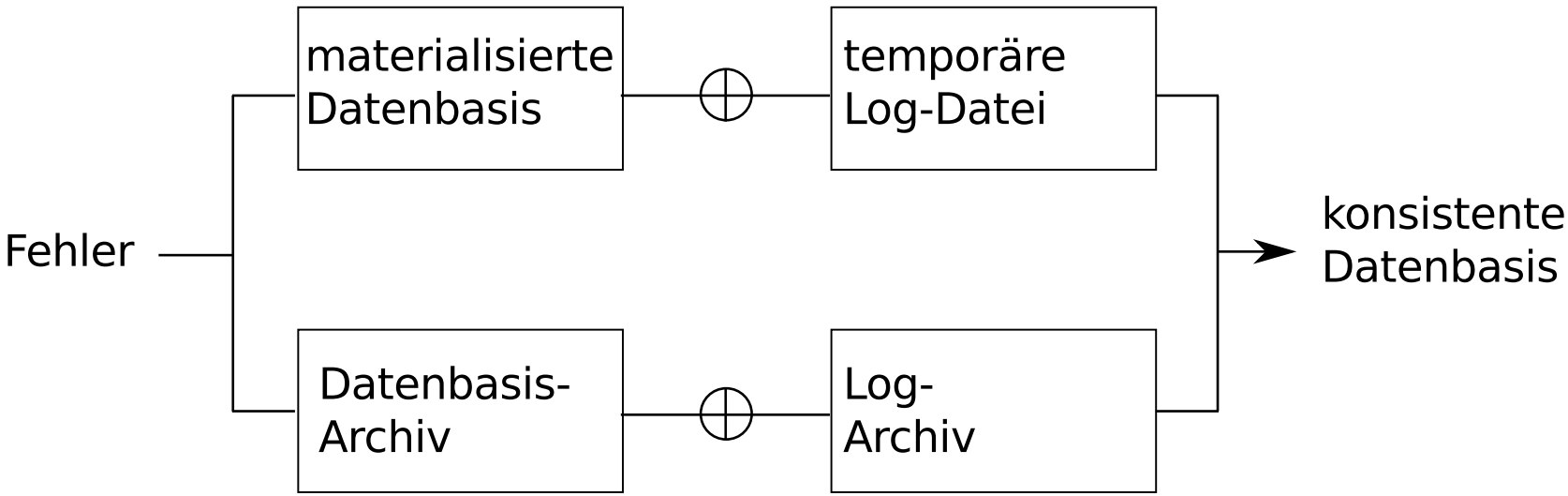
- **MinDirtyPageLSN:**
  - Die minimale LSN, deren Änderungen noch nicht ausgeschrieben wurde
  - Die älteste Änderungsoperation, die eine Seite geändert hat (hat "dreckig" werden lassen).
  - Die älteste Änderung, die wiederholt werden muss (im Redo)
- **MinLSN:**
  - Die kleinste LSN der zum Sicherungszeitpunkt aktiven TA
  - Erste/älteste Änderungsoperation aller Loser-TA
  - Die älteste Änderung, die rückgängig gemacht werden muss (im Undo)

Zusammenfassung der drei Arten von Sicherungspunkten



Recovery

Recovery nach einem Verlust der materialisierten Datenbasis.



Dies kann auch auf einzelne Seiten angewandt werden z.B. bei einzelnen fehlerhaften Seiten (Media Recovery)

Zusammenfassung Recovery

- Motivation: ACID (Speziell Atomicity und Durability)
- Aber auch: High Availability (durch kleine MTTR), also Effizienz
- Es gibt verschiedene Arten von Recovery (je nach Fehler)
- Haben (hauptsächlich) über Crash-Recovery gesprochen
- Essenz: Speichern von Log-Informationen
- WAL-Prinzip und Commit-Regel
- Redo von Gewinner-TA, Undo von Verlierer-TA
- Idempotenz des Wiederanlaufs
- Sicherungspunkte zum schnelleren Wiederanlauf (nicht das ganze Log betrachten müssen)
- Kurz: Zurücksetzen von Transaktionen