Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Implikationen von ACID auf Anforderungen zu Recovery

Durability

- Änderungen an der Datenbank, die durch erfolgreich(!) abgeschlossene (d.h. committed) Transaktionen verursacht wurden, müssen dauerhaft gespeichert sein.
- D.h. bei einem Crash der DB muss nach Wiederanlauf geschaut werden, ob dies tatsächlich der Fall ist.

Atomicity

- Falls eine Transaktion noch nicht erfolgreich abgeschlossen wurde und ein DB-Crash auftritt, muss beim Wiederanlauf darauf geachtet werden, dass etwaige Änderungen in der Datenbasis rückgängig gemacht werden.

Fehlerklassifikation

Lokaler Fehler in einer noch nicht festgeschriebenen (committed) Transaktion:

- Wirkung der TA muss zurückgesetzt werden

Fehler mit Hauptspeicherverlust:

- Abgeschlossene TAs müssen erhalten bleiben
- Noch nicht abgeschlossene TAs müssen zurückgesetzt werden

Fehler mit Hintergrundspeicherverlust:

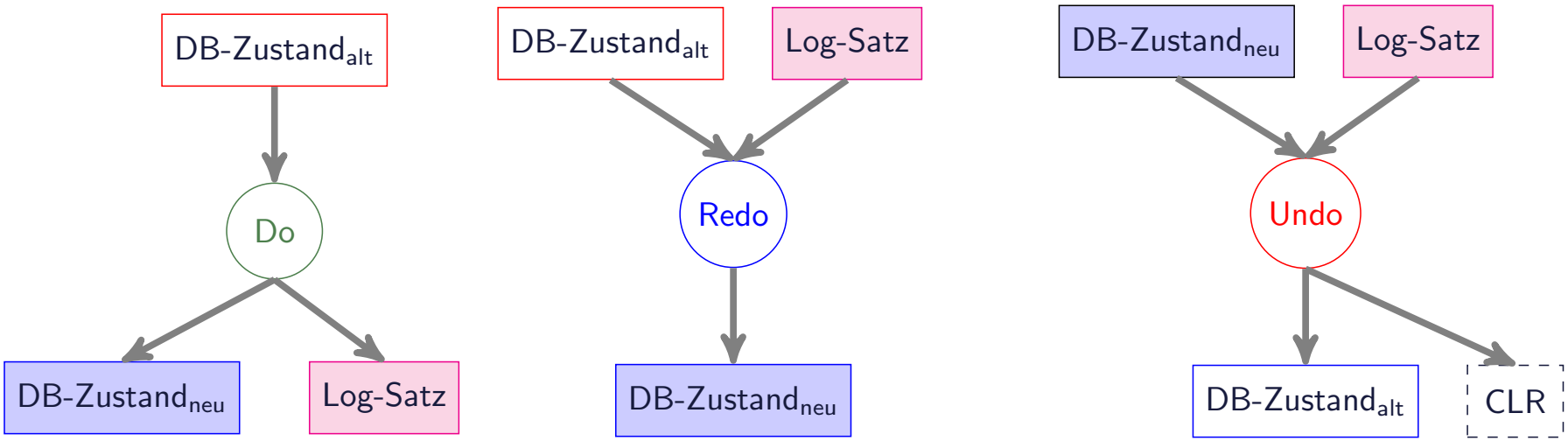
- Archiv einspielen

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



Recovery-Oriented Computing

Systemverfügbarkeit A (availability)

- MTTF: Mean Time To Failure
- MTTR: Mean Time To Repair

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

Warum Recovery-Oriented Computing?

- Hardwarefehler, Softwarefehler passieren (sind nicht vermeidbar) und müssen adressiert werden.
- Lange Systemausfälle sind sehr sichtbar (z.B. Amazon, Facebook, Ebay!)

Number of Nines & Entwicklungsziele

Availablility wird manchmal beschrieben in Anzahl von Neunen (Nines), wie in "five nines", oder 99.999%. 99.99% entspricht ungefähr 1 Minute Ausfall in einer Woche, bzw. circa 50 Minuten Ausfall in einem Jahr.

- "Build a system used by millions of people that is always available – out less than 1 second per 100 years = 8 9's of availability" (J. Gray: 1998 Turing Award Lecture)

Jim Gray: We have added three 9s in 45 years (starting with 90%), or about 15 years per order-of-magnitude improvement in availability. We should aim for five more 9s: an expectation of one second outage in a century. This is an extreme goal, but it seems achievable if hardware is very cheap and bandwidth is very high. One can replicate the services in many places, use transactions to manage the data consistency, use design diversity to avoid common mode failures, and quickly repair nodes when they fail. Again, this is not something you will be able to test: so achieving this goal will require careful analysis and proof.

Wie kann man annährend $A = 1,0$ erreichen?

- MTTF $\rightarrow \infty$?
- MTTR $<<$ MTTF!

Es gibt Fehler die man nur sehr schwer oder gar nicht nicht durch testen in den Griff bekommen kann. Sie treten nichtdeterministisch auf, oft aufgrund von Nebenläufigkeit in Threads, unter hoher Last, oder in anderen seltenen Fällen.

Solche Probleme werden auch "Heisenbugs" genannt (Jim Gray); nach Heisenbergs Unschärferelation.

D.h. ein schneller Wiederanlauf (Recovery) ist essentiell für hohe Verfügbarkeit (Availability); da diese "Heisenbugs" die MTTF in der Praxis einschränken.

Grundlagen der DB-Recovery

Aufgabe des DBMS

- Automatische Behandlung aller erwarteten Fehler

Was sind erwartete Fehler?

- DB-Operation wird zurückgewiesen, Commit wird nicht akzeptiert, ...
- Stromausfall, DBMS-Probleme
- Geräte funktionieren nicht (Spur, Zylinder, Platte defekt)
- Beliebiges Fehlverhalten der Gerätesteuerung
- ...

Fehlermodelle von (zentralisierten) DBMS

- Transaktionsfehler
- Systemfehler
- Gerätefehler
- Katastrophen

Recovery-Arten

1. Transaktions-Recovery
   - Zurücksetzen einzelner (noch nicht abgeschlossener) TA im laufenden Betrieb (TA-Fehler, Deadlock, etc.)

   → Vollständiges Zurücksetzen auf BOT (TA-Undo)

   → Partielles Zurücksetzen auf Rücksetzpunkt (Savepoint) innerhalb der Transaktion
2. Crash-Recovery nach Systemfehler
   - Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustands:

   → (partielles) Redo für erfolgreiche TA (Wiederholung verlorengegangener Änderungen

   → Undo aller durch Ausfall unterbrochenen TA (Entfernung der Änderungen aus der DB)

Recovery-Arten (2)

3. Medien-Recovery nach Gerätefehler
   - Spiegelplatten, bzw.
   - vollständiges Wiederholen (Redo) aller Änderungen auf einer Archivkopie
4. Katastrophen-Recovery
   - Nutzung einer aktuellen DB-Kopie in einem "entfernten" System oder
   - stark verzögerte Fortsetzung der DB-Verarbeitung mit repariertem/neuem System auf Basis gesicherter Archivkopien

Crash-Recovery

Im folgenden wird Crash-Recovery betrachtet.

Idee/Ziel

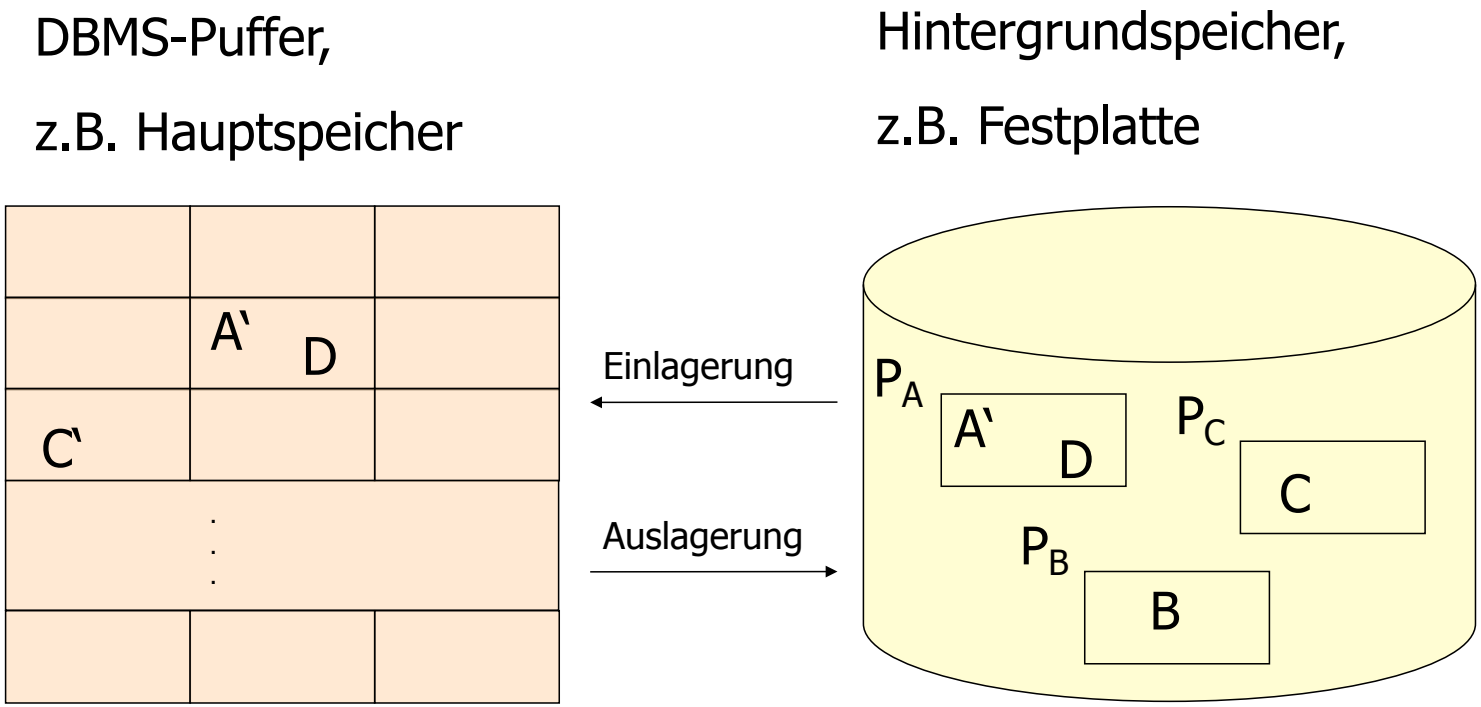
- Wiederanlauf (Restart) des DBS nach einem Systemfehler
- Ziel: Nach Wiederanlauf den jüngsten transaktionskonsistenzen DB-Zustand wiederherstellen, der zum Fehlerzeitpunkt gültig war.
- Dazu wird materialisierte Datenbasis und Log-Datei benutzt.
- Zusätzliche Anforderung: Idempotenz! D.h. bei mehrfacher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



Einbringungsstrategie

Update in Place:

- jede Seite hat genau eine "Heimat" auf dem Hintergrundspeicher
- der alte Zustand einer Seite wird überschrieben

Twin-Block-Verfahren:

- Jede Seite hat zwei Versionen
- Anordnung für Seiten P_A, P_B und P_C

P_A^0	P_A^1	P_B^0	P_B^1	P_C^0	P_C^1	...

Schattenspeicherkonzept:

- nur geänderte Seiten werden dupliziert
- weniger Redundanz als beim Twin-Block-Verfahren

Die Speicherhierarchie

Ersetzung von Puffer-Seiten:

- ¬**steal:** Ersetzung von Seiten, die von einer noch aktiven Transaktion modifiziert wurden, ausgeschlossen ⇒ "dreckige" Seiten (dirty pages) müssen im Puffer bleiben.
- **steal:** Jede nicht fixierte Seite ist prinzipiell ein Kandidat für die Ersetzung, falls neue Seiten eingelagert werden müssen, auch "dreckige" Seiten.

Einbringen von Änderungen abgeschlossener TAs:

- **force:** Änderungen werden bei commit auf den Hintergrundspeicher geschrieben (flush).
- ¬**force:** geänderte Seiten können auch nach commit im Puffer verbleiben.

dirty page = Inhalt der Seite im HS ≠ Inhalt der Seite auf FS

Auswirkungen auf Recovery

Undo: entfernt ungültige Einträge aus der DB.

Redo: fügt gültige Einträge in die DB sein.

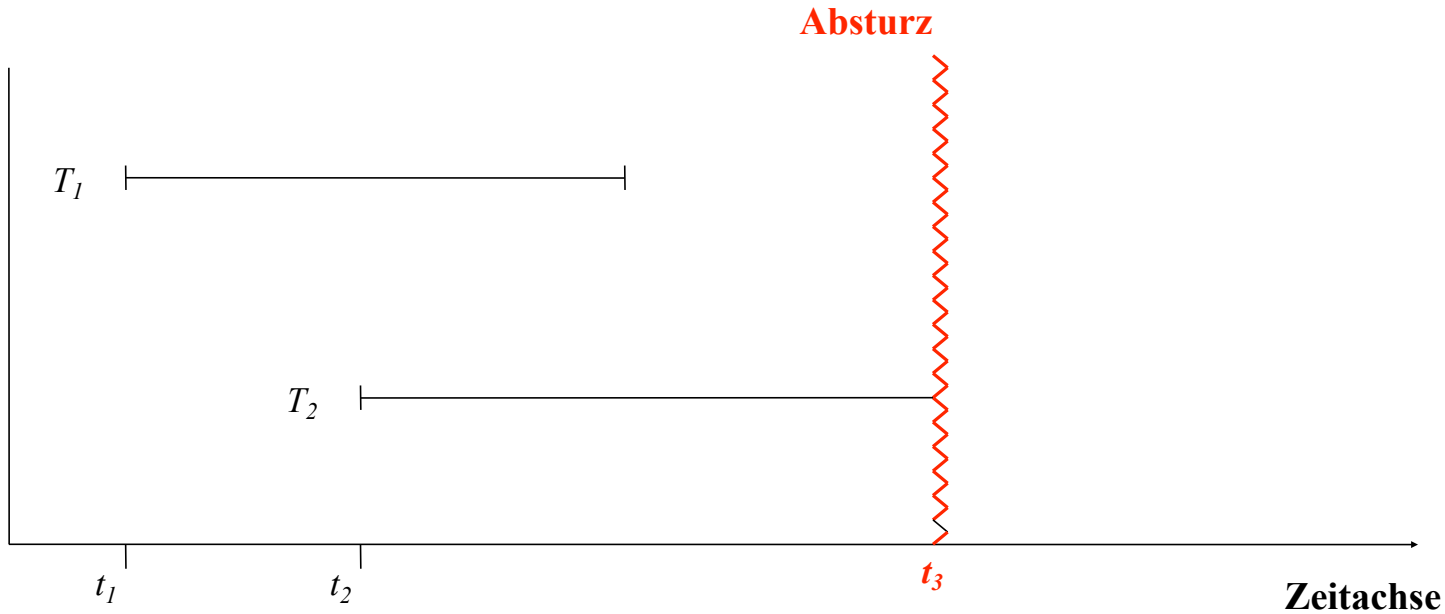
	force	**¬force**
¬steal	• kein Undo • kein Redo ⇒ **keine** Recovery	• Redo • kein Undo
steal	• kein Redo • Undo	• Redo • Undo

- Was ist besser? steal oder ¬steal?
- Was ist besser? force oder ¬force?
- Annahme: Schreiboperationen von Seiten müssen atomar sein.

Hier zugrunde gelegte Systemkonfiguration

- **steal**
  "dreckige Seiten" können in die Datenbank (auf Platte) geschrieben werden.
- **¬force**
  geänderte Seiten sind **möglicherweise** noch nicht auf die Platte geschrieben
- **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.

Wiederanlauf nach einem Fehler



- Transaktionen der Art T_1 müssen hinsichtlich ihrer Wirkung vollständig nachvollzogen werden. Diese TAs nennt man **Winner**.
- Transaktionen, wie wie T_2 zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden. Diese TAs nennt man **Loser**.

Transaktionsparadigma: Was muss bei Fehlern garantiert werden?

Pufferinhalt geht verloren, was dann?

Undo

Alle durch nicht abgeschlossene Transaktionen schon in die materialisierte Datenbasis eingebrachten Änderungen müssen rückgängig gemacht werden.

Redo

Alle noch nicht in die materialisierte Datenbasis eingebrachten Änderungen durch abgeschlossene Transaktionen müssen nachvollzogen werden.

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

Write-Ahead-log-Prinzip (WAL)

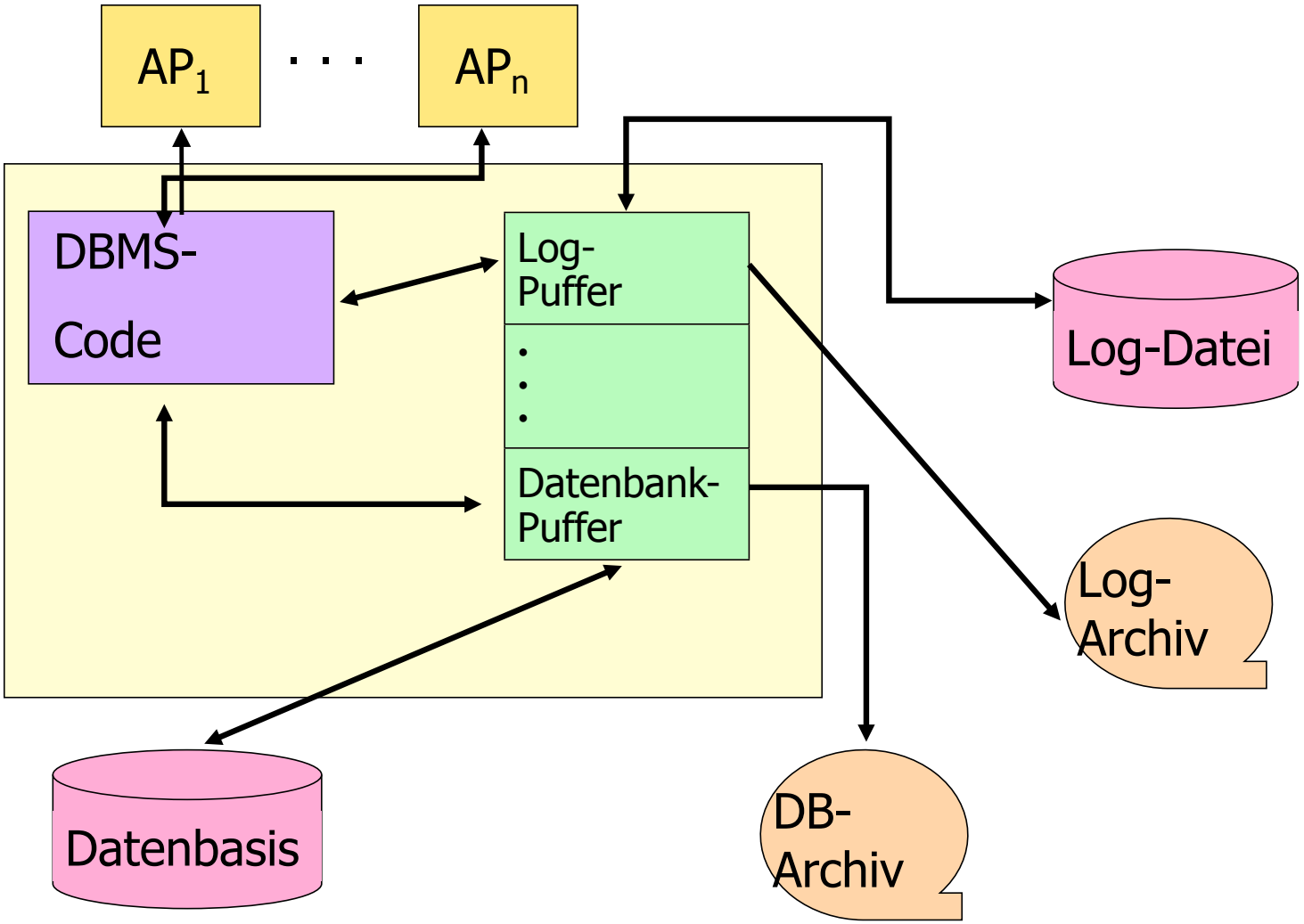
Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in die Log-Datei und das Log-Archiv ausgeschrieben werden.

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

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

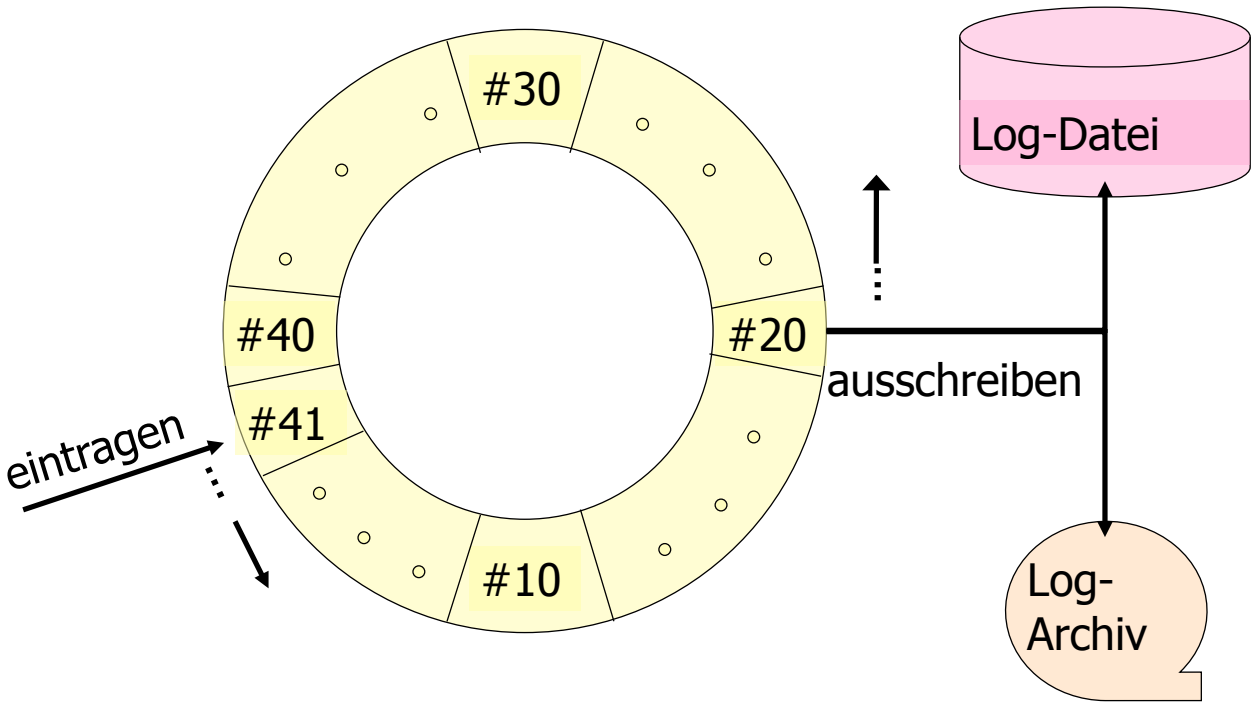
C. Mohan et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.
`http://www.cs.berkeley.edu/~brewer/cs262/Aries.pdf`

Schreiben von Log-Informationen



- Log-Informationen werden zweimal geschrieben: Log-Datei für schnellen Zugriff und Log-Archiv

Anordnung des Log-Ringpuffers



- Kontinuierliches Ausschreiben
- Aber Achtung: WAL und Commit-Regel beachten!
- Log-Puffer ist normalerweise kleiner als DB-Puffer
- Groß genug um i.d.R. laufende Transaktionen zu enthalten, so dass beim Rücksetzen einer TA dies anhand des Puffers gemacht werden kann.

Protokollierung von Änderungsoperationen

Struktur der Log-Records

[LSN, TransaktionsID, PageID, Redo, Undo, PrevLSN]

LSN (Log Sequence Number)

- eine eindeutige Kennung des Log-Records
- LSNs müssen monoton aufsteigend vergeben werden,
- die chonologische Reihenfolge der Protokolleinträge kann dadurch ermittelt werden.
- z.B. Offset des Log-Records im Log-File

TransaktionsID

- die ID der Transaktion, die die Änderung durchgeführt hat.

PageID

- die ID der Seite, auf der die Änderungsoperation vollzogen wurde
- Wenn eine Änderung mehr als eine Seite betrifft, müssen entsprechend viele Log-Records generiert werden.

Protokollierung von Änderungsoperationen

Struktur der Log-Records

[LSN, TransaktionsID, PageID, Redo, Undo, PrevLSN]

- Die **Redo**-Information gibt an, wie die Änderung nachvollzogen werden kann.
- Die **Undo**-Information beschreibt, wie die Änderung rückgängig gemacht werden kann.
- **PrevLSN** ist ein Zeiger auf den vorhergehenden Log-Record der jeweiligen Transaktion. Diesen Zeiger benötigt man aus Effizienzgründen.

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]

Logische oder physische Protokollierung?

Physische Protokollierung

Es werden Inhalte/Zustände protokolliert:

1. **before-image** enthält den Zustand vor Ausführung der Operation, z.B. als byte-array
2. **after-image** enthält den Zustand nach Ausführung der Operation, z.B. als byte-array

Logische Protokollierung

- Zustandsübergang wird protokolliert, z.B. $a = a + 10$

Seiten-LSN (PageLSN)

Speicherung der Seiten-LSN (PageLSN)

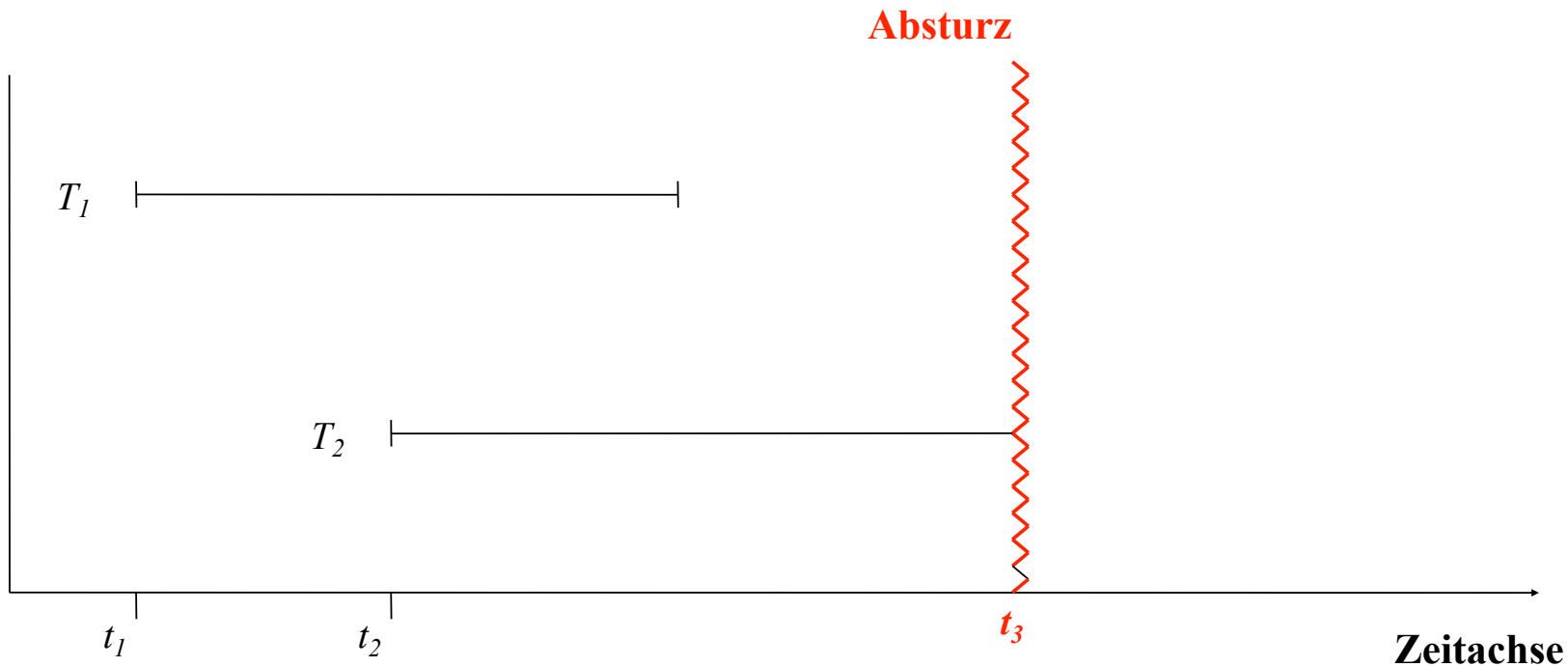
- Die "Herausforderung" besteht darin, beim Wiederanlauf zu entscheiden, welche Version diese Seite hat.
- Dazu wird auf jeder Seite die LSN des jüngsten diese Seite betreffenden Log-Eintrags gespeichert.

LSN zur Erkennung ob Before oder AfterImage

LSN des Log-Eintrags wird mitkopiert, 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 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.

Wiederanlauf nach einem Fehler



- Transaktionen der Art T_1 müssen hinsichtlich ihrer Wirkung vollständig nachvollzogen werden. Diese TAs nennt man **Winner**.
- Transaktionen, wie wie T_2 zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden. Diese TAs nennt man **Loser**.

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.

Redo und PageLSN

- Log-Satz einer Änderung bezieht sich auf genau eine DB-Seite
- Die im Seitenkopf gespeicherte PageLSN entspricht der LSH des Log-Eintrags, welcher die zuletzt auf der Seite ausgeführte Änderung protokolliert.
- Eine Änderung, deren Log-Eintrag eine LSN aufweist, die kleiner oder gleich der PageLSN ist, befindet sich somit bereits in der Seite und braucht nicht wiederholt zu werden!
- Ansonsten muss Redo ausgeführt werden.

Für Log-Eintrag L und Seite B:

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

Achtung: Undo wird immer ausgeführt, egal welche LSN in der Seite steht. Weil entweder wurde Änderungs-Aktion ausgeführt vor Crash, oder beim Redo!

Selektives vs. vollständiges Redo

- Selektives Redo: Nur Winner TA werden im Redo berücksichtigt.
- Vollständiges Redo: Alle TA (also auch die Loser) werden im Redo berücksichtigt. (Dies betrachten wir hier in der VL).

Verwendung der PageLSN-Werte zur Redo-Recovery erfolgt beim selektiven wie beim vollständigen Redo, wie beschrieben.

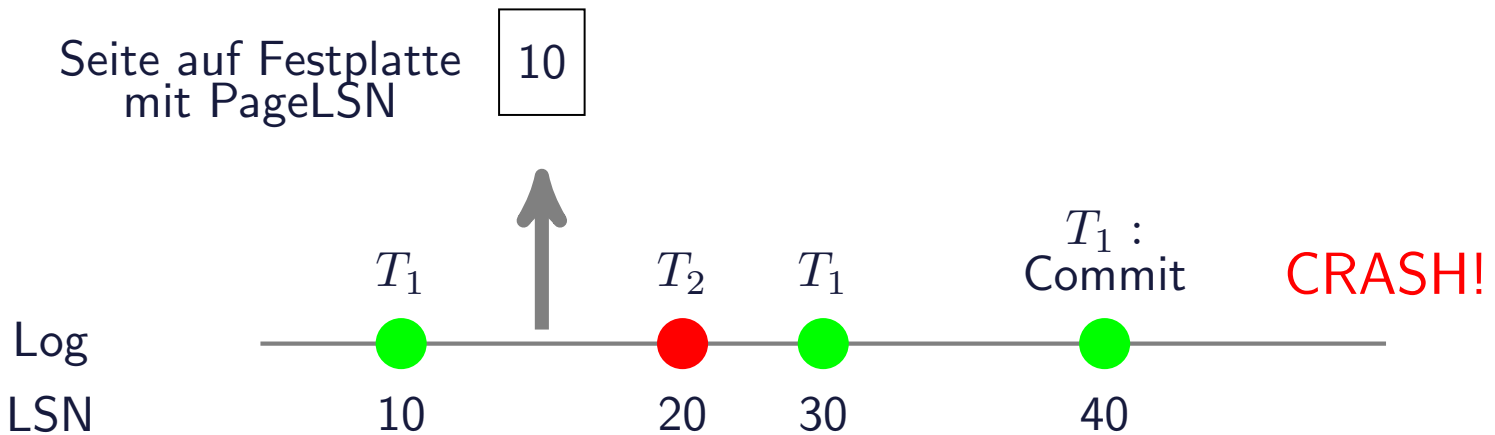
Unterschied bei Redo bzw. Undo der Verlierer-Transaktionen

- Selektives Redo funktioniert nur bei Seiten-Sperren
- Wir betrachten hier aber die Verwendung von Satzsperren!

Selektives vs. vollständiges Redo (2)

Angenommen: Satzsperren und selektives Redo. Was geht schief?

- T_1 und T_2 verändern unabhängig verschiedene Sätze in derselben Seite.
- Beim selektiven Redo werden nur Änderungen der Gewinner-Transaktion T_1 wiederholt.
- Also wird PageLSN von 10 auf 30 erhöht.
- Im Undo-Lauf wird dann die Änderung von T_2 zurückgesetzt, da aufgrund der PageLSN 30 davon ausgegangen wird, dann Änderung 20 in der Seite enthalten ist (ist sie aber nicht!).



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)

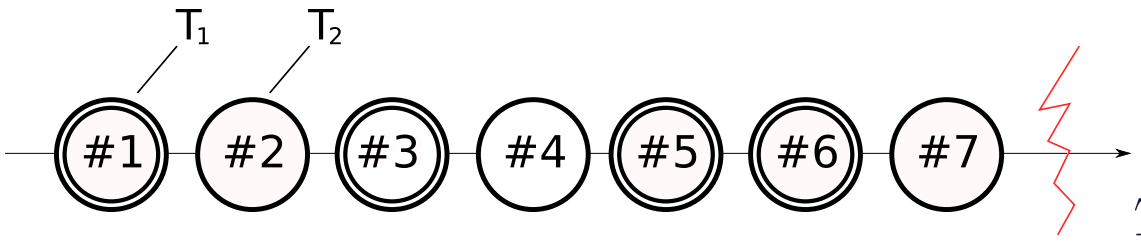
Der Redo Fall:

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

Der Undo Fall:

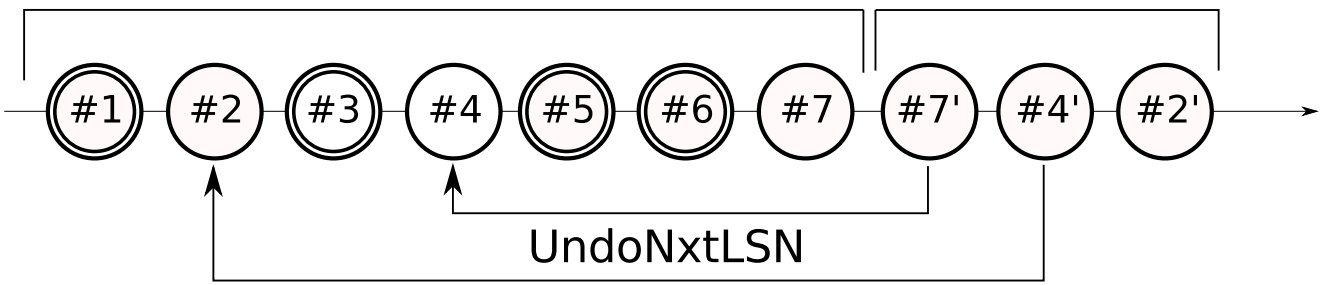
- Kompensations-Protokolleinträge (**CLR**, compensation log record)
- Für jede Undo-Operation wird ein CLR angelegt

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

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?

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>

Einschub: Lokales Zurücksetzen einer Transaktion

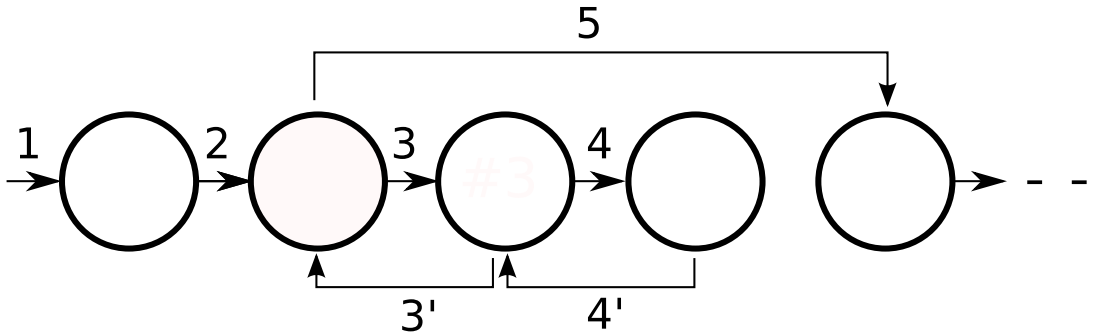
Isoliertes Zurïcksetzen einer einzelnen Transaktion

- 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 zurïcklaufen
- Und Undo-Operationen ausführen
- Aber: Vorher noch mit Compensation Log Record protokollieren.

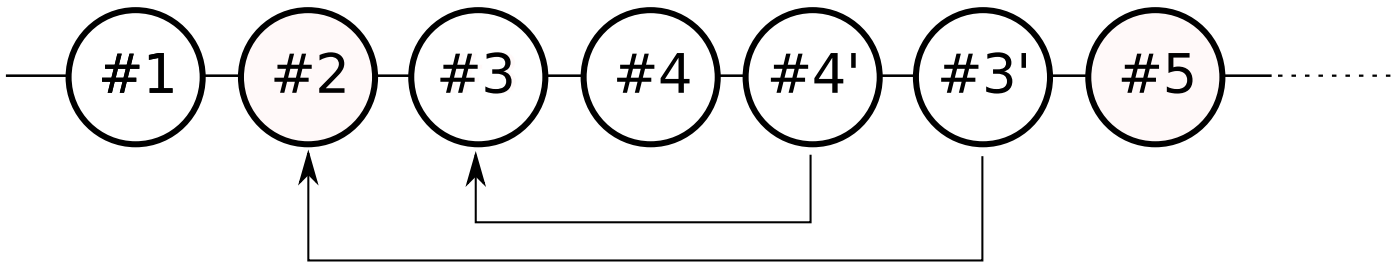
Einschub: Lokales Zurücksetzen einer Transaktion (2)

Z.B. partielles Zurücksetzen einer Transaktion

Historie



Log



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

Sicherungspunkte (=checkpoints)

Und zurück zu Crash-Recovery ....

Beobachtung

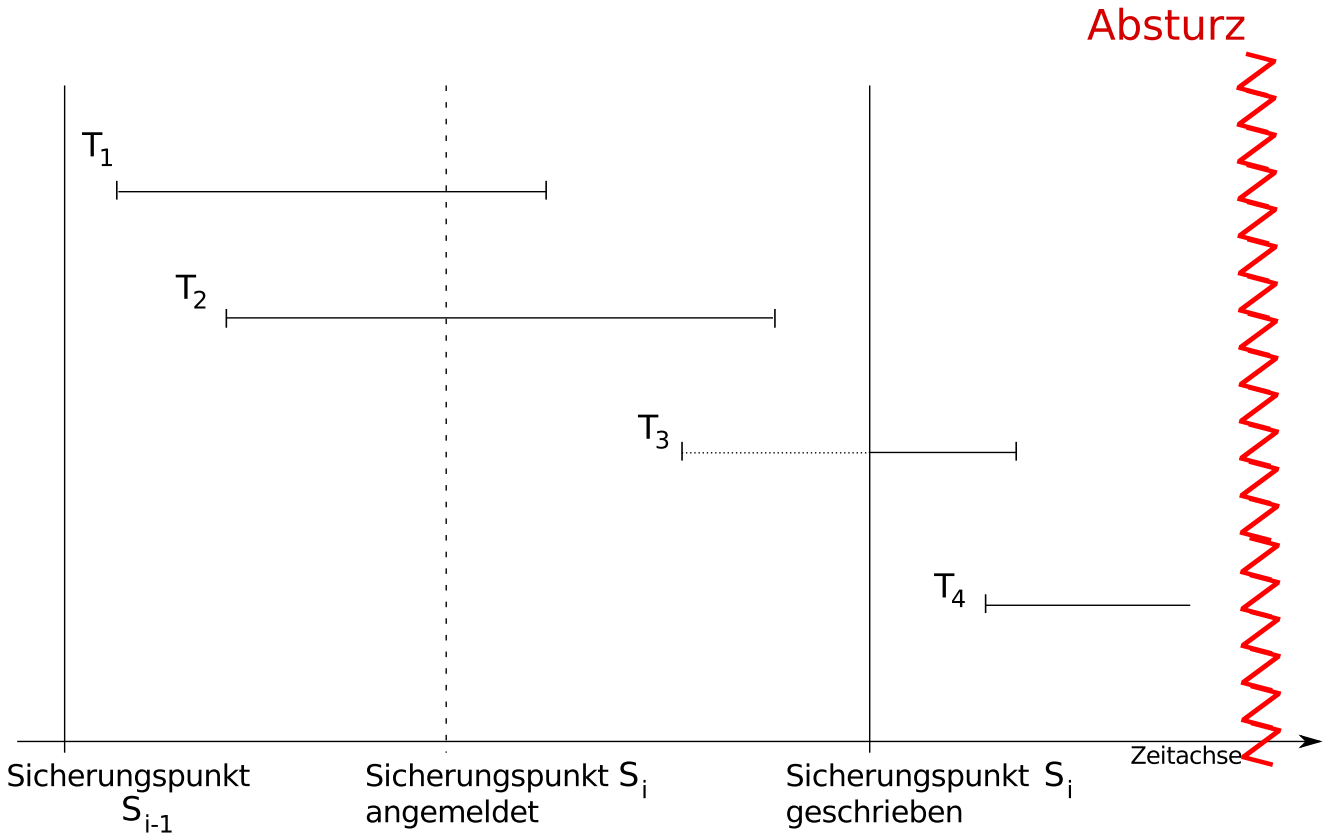
- Mit zunehmender Betriebszeit des Datenbanksystems wird Wiederanlauf immer langwieriger, da die Log-Datei immer umfangreicher wird.

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)

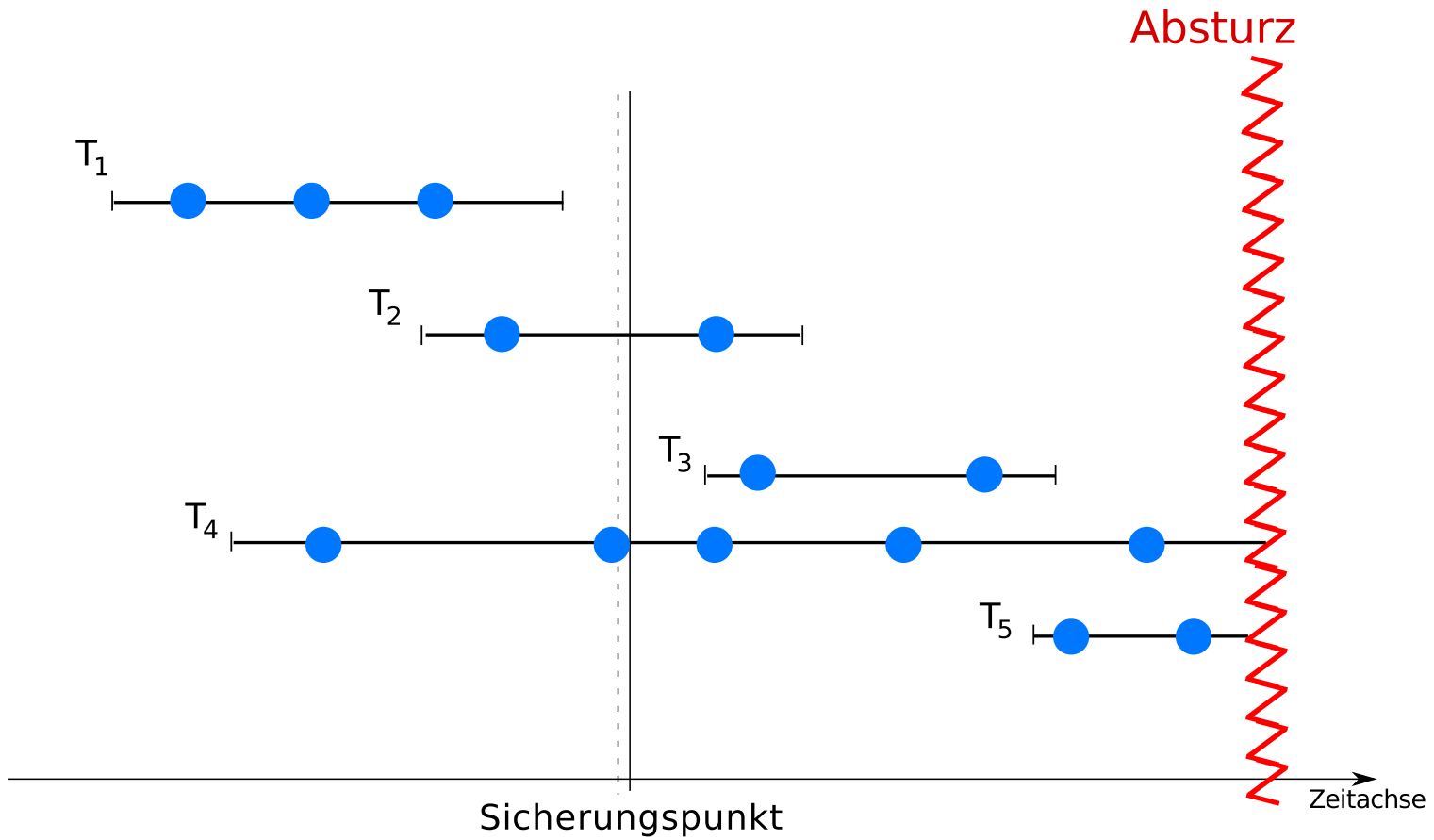
- Neu ankommende TA müssen warten. Laufende TA werden zu Ende ausgeführt.
- Dann schreiben aller modifizierten Seiten auf Hintergrundspeicher.



Achtung: checkpoint ≠ savepoint (Terminologie)

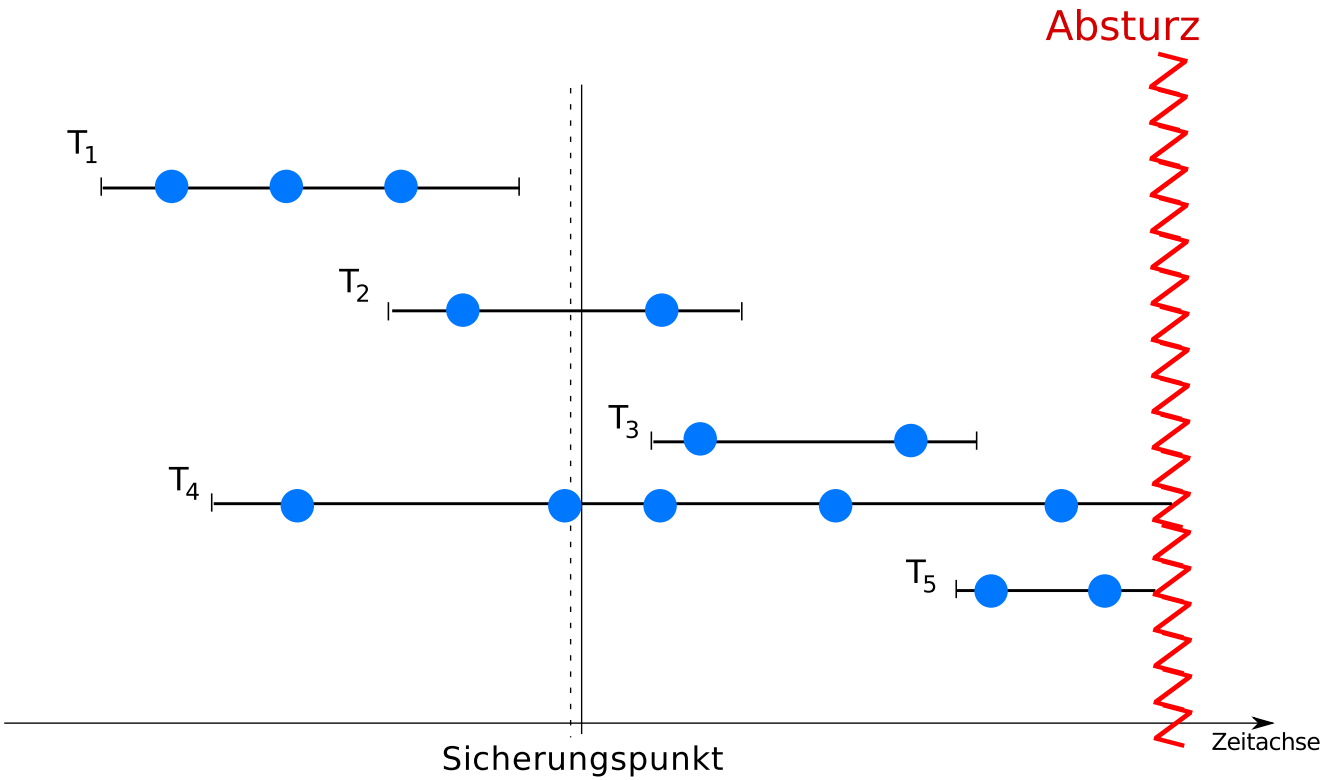
Aktionskonsistente Sicherungspunkte

- 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



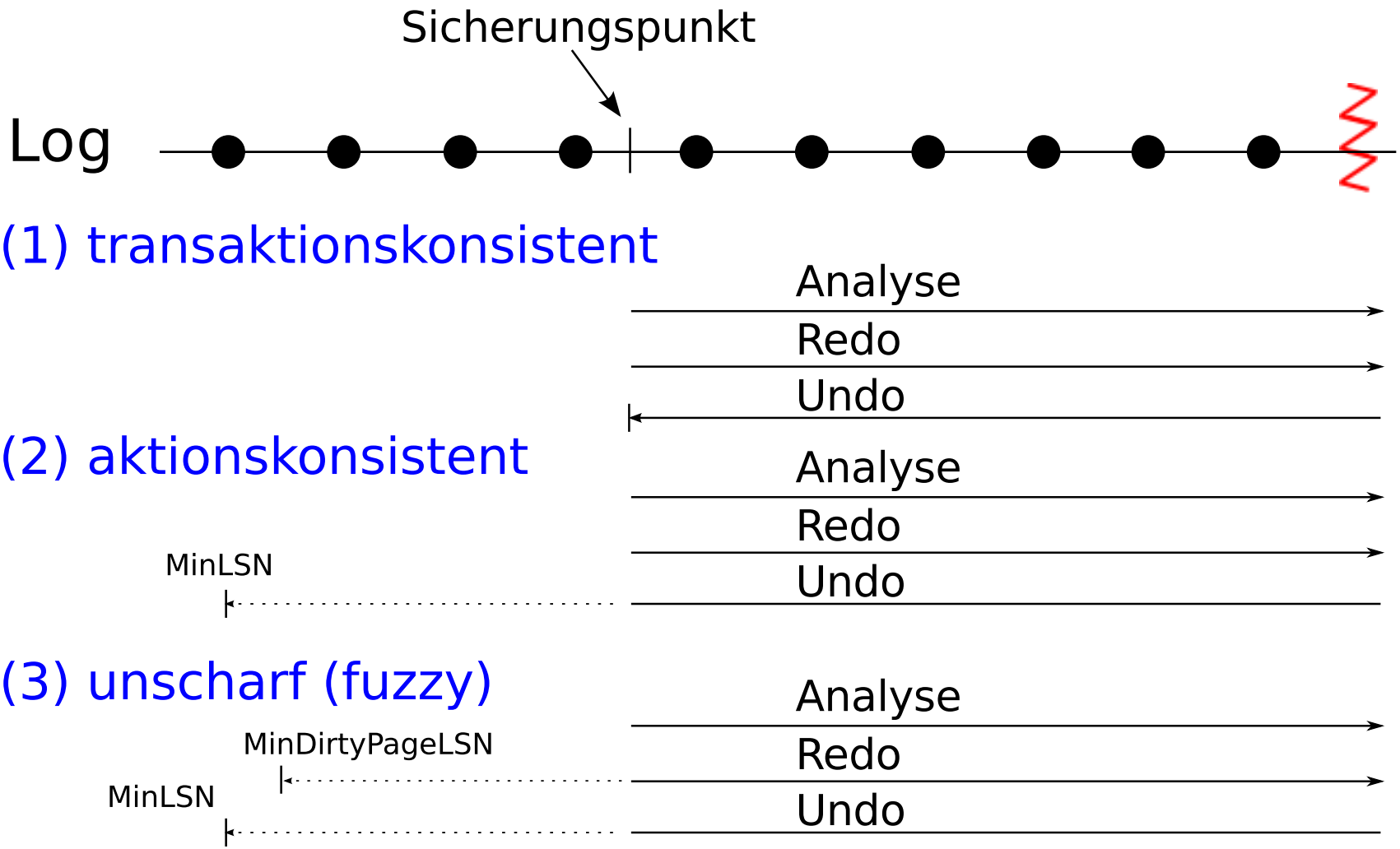
Unscharfe (fuzzy) Sicherungspunkte

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

Terminologie:

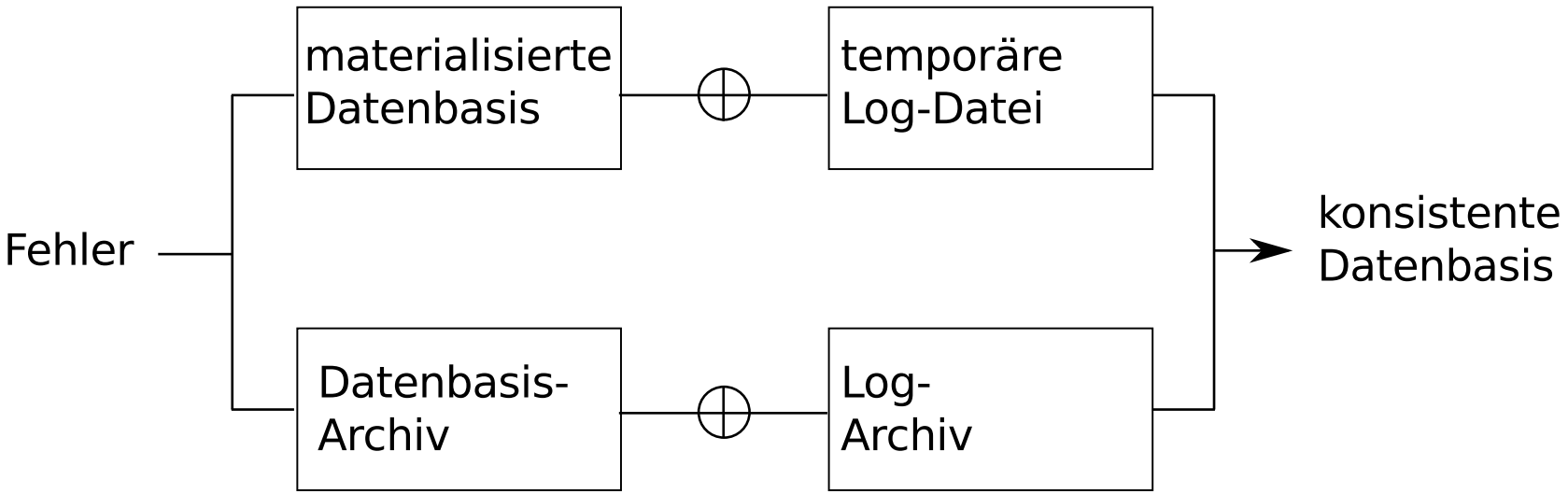
- **MinDirtyPageLSN:**
  - Die minimale LSN, deren Änderungen noch nicht ausgeschrieben wurde
  - Die älteste Änderungsoperation, die eine Seite geändert hat (hat "dreckig" werden lassen).
  - Die älteste Änderung, die wiederholt werden muss (im Redo)
- **MinLSN:**
  - Die kleinste LSN der zum Sicherungszeitpunkt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