9. Logging und Recovery[1]

- **DB-Recovery**
  - Anforderungen und Begriffe
  - Fehler- und Recovery-Arten
- **Logging-Verfahren**
  - Klassifikation und Bewertung
  - Aufbau der Log-Datei, Nutzung von LSNs
- **Abhängigkeiten zu anderen Systemkomponenten**
  - Externspeicherabbildung: Einbringstrategie
  - Zusammenspiel mit der DB-Puffer- und Sperrverwaltung
- **Commit-Behandlung** (Gruppen-, Prä-Commit)
- **Sicherungspunkte**
  Direkte und unscharfe Sicherungspunkte (*Checkpoints*)
- **Klassifikation von DB-Recovery-Verfahren**
- **Crash-Recovery**
  - Allgemeine Restart-Prozedur
  - Restart-Bespiel (Selektives Redo)
  - Einsatz von Compensation Log Records
  - Restart-Beispiel (Repeating History)
- **Transaktions-Recovery**
- **Die Zehn Gebote**
- **Medien-Recovery**

---

1. Härder, T., Reuter, A.: Principles of Transaction Oriented Database Recovery, in: ACM Computing Surveys 15:4, Dec. 1983, 287-317.

Recovery-Oriented Computing

- **Systemverfügbarkeit A**
  - MTTF: Mean Time To Failure, MTTR: Mean Time To Repair

    A = MTTF / (MTTF + MTTR)

  - Wie erreicht man annähernd A = 1,0 ?
    - **MTTF $\rightarrow \infty$?**
    - **MTTR << MTTF !**

- **Warum ROC?**
  - Falsche Operateur-Aktionen sowie HW- und SW-Fehler sind Tatsachen, mit denen man fertig werden muss, und keine Probleme, die zu lösen sind[2]
  - Lange Systemausfälle sind sehr sichtbar (siehe Ebay-Ausfall von 280 min.)!
  - MTTR soll minimiert und kann direkt gemessen werden (MTTF von Magnetplatten ist heute 120 Jahre)
  - Verkürzung der MTTR (auf Anwendungsebene) verbessert die Benutzererfahrung, was das Systemverhalten betrifft
  - Schnelle und häufige „Recovery“ (auf systeminternen Ebenen/in Komponenten) kann die effektive MTTF verlängern (Verjüngungseffekt!)

    ➥ ***Der Fokus liegt auf ROC!***[3]

---

2. If a problem has no solution, it may not be a problem but a fact, not to be solved but to be coped with over time (Shimon Peres)
3. Beobachtung: Welche Konsequenzen sind aus ROC zu ziehen?
   Die System- einschl. Betriebskosten von DB-Systemen sind 3- bis 18-mal höher als der Kaufpreis der HW (Cluster-basierte Systeme) und 1/3 bis 1/2 dieser Kosten wird für Recovery oder für Fehlervorsorge aufgewendet

Grundlagen der DB-Recovery

- **Aufgabe des DBMS:**
  Automatische Behandlung aller erwarteten Fehler

- **Was sind erwartete Fehler?**[4]
  - DB-Operation wird zurückgewiesen, Commit wird nicht akzeptiert, . . .
  - Stromausfall, DBMS-Probleme, . . .
  - Geräte funktionieren nicht (Spur, Zylinder, Platte defekt)
  - auch beliebiges Fehlverhalten der Gerätesteuerung?
  - falsche Korrektur von Lesefehlern? . . .

- **Was sind die Besonderheiten der DBS-Fehlerbehandlung?**
  - Begrenzung und Behebung der zur Laufzeit möglichen Fehler
    (wie auch bei anderen fehlertoleranten Systemen)
  - „Reparatur" der statischen Struktur der DB

- **Allgemeine Probleme**
  - Fehlererkennung
  - Fehlereingrenzung
  - Abschätzung des Schadens
  - Durchführung der Recovery

- **Fehlermodell von zentralisierten DBMS**
  - Transaktionsfehler
  - Systemfehler
  - Gerätefehler
  - Katastrophen

---

4. Kommerzielle Anwendungen auf Großrechnern sind durch ihre Zuverlässigkeit gekennzeichnet. Nicht selten besteht der Code bis zu 90% aus (erprobten) Recovery-Routinen (W. G. Spruth).

Grundlagen der DB-Recovery (2)

- **Voraussetzung:**
  Sammeln redundanter Informationen während des Normalbetriebs

- **Welcher Zielzustand soll erreicht werden?**
  - früher: beliebiger Abbruch der DB-Verarbeitung
  - Verbesserung: Sicherungspunkte bei „Langläufern”

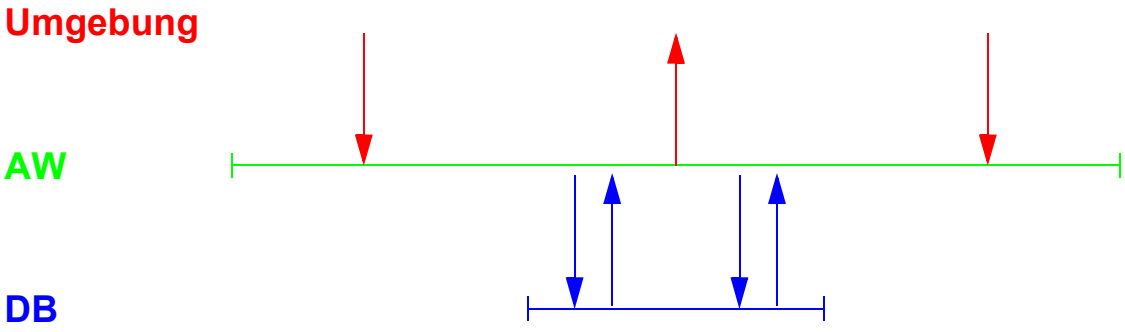
- **Transaktionsparadigma verlangt:**
  - Alles-oder-Nichts-Eigenschaft von Transaktionen
  - Dauerhaftigkeit erfolgreicher Änderungen

- **Zielzustand nach erfolgreicher Recovery:**

  Durch die Recovery-Aktionen ist der jüngste Zustand vor Erkennen des Fehlers wiederherzustellen, der allen semantischen Integritäts-bedingungen entspricht, der also ein möglichst aktuelles, exaktes Bild der Miniwelt darstellt

  ➥ ***jüngster transaktionskonsistenter DB-Zustand!***

- **In welchem Zustand befindet sich die Systemumgebung?**
  (Betriebssystem, Anwendungssystem, andere Komponenten)



Grundlagen der DB-Recovery (3)

- **Wie soll Recovery durchgeführt werden?**

- **Forward-Recovery**
  - Non-Stop-Paradigma (Prozesspaare usw.)
  - Fehlerursache häufig falsche Programme, Eingabefehler u. ä.
  - durch Fehler unterbrochene TA sind zurückzusetzen

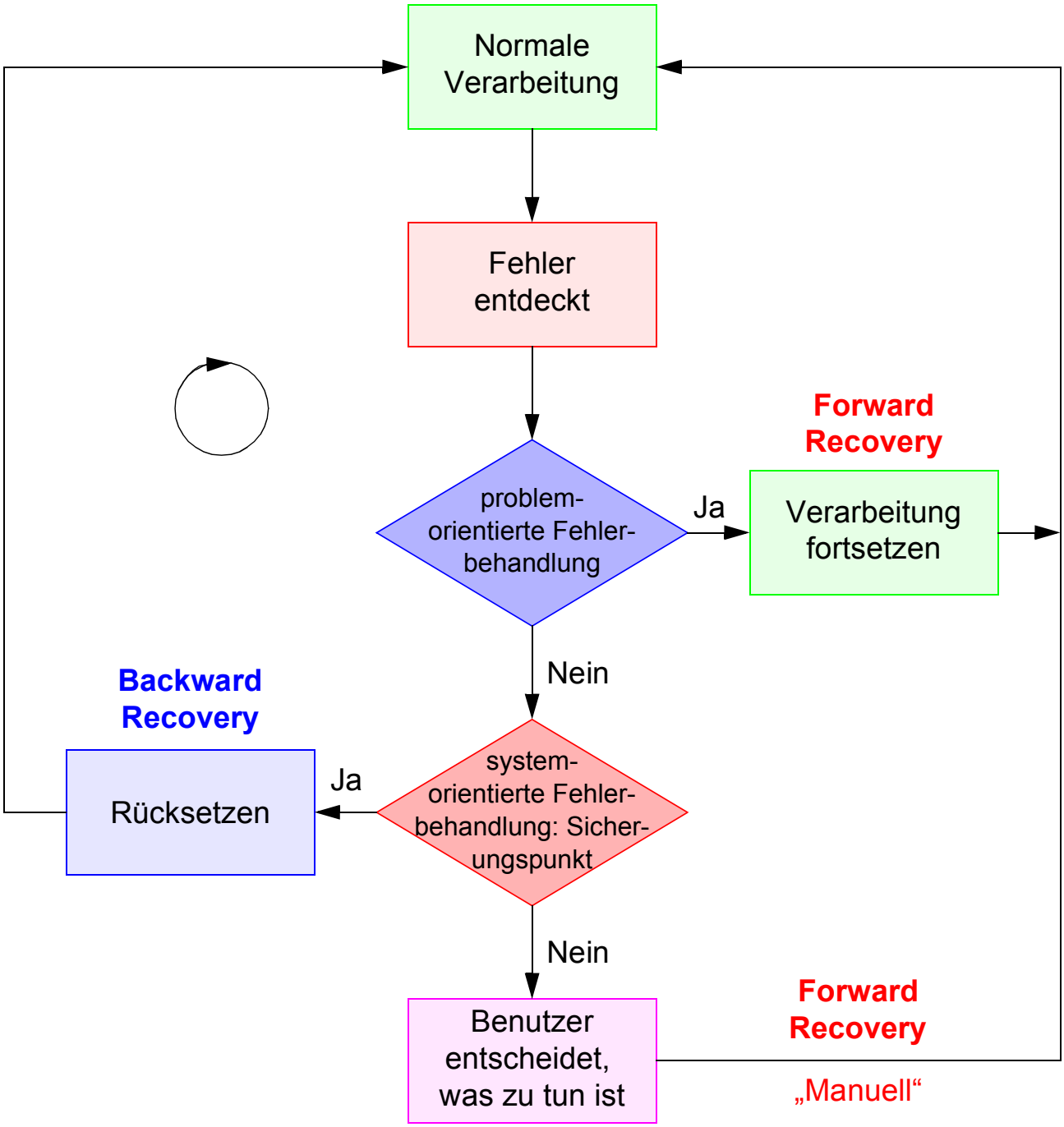
  ➥ Forward-Recovery i. Allg. nicht anwendbar!

- **Backward-Recovery**
  - setzt voraus, dass auf allen Abstraktionsebenen genau definiert ist, auf welchen Zustand die DB im Fehlerfall zurückzusetzen ist
  - Zurücksetzen auf konsistenten Zustand und Wiederholung
  - Warum funktioniert Backward-Recovery?
    (Unterscheidung von „Bohrbugs" und „Heisenbugs")

- **"A recoverable action is 30% harder and requires 20% more code than a non-recoverable action" (J. Gray)**
  - Anweisungs- und TA-Atomarität gefordert
  - Zwei Prinzipien der Anweisungs-Atomarität möglich
    - **„Do things twice"**
      (vorbereitende Durchführung der Operation; wenn alles OK, erneuter Zugriff und Änderung)
    - **„Do things once"**
      (sofortiges Durchführen der Änderung; wenn Fehler auftritt, internes Zurücksetzen)
  - Zweites Prinzip wird häufiger genutzt (ist optimistischer und effizienter)

Recovery – Begriffsklärung

- **Grundsätzliche Vorgehensweisen**



- **Was passiert, wenn**
  - nach Backward-Recovery der Fehler nicht behoben ist?
  - nach Forward-Recovery die „normale Verarbeitung“ weitergeführt bzw. wieder aufgenommen wird?

Fehlerarten

Auswirkung eines Fehlers auf	Fehlertyp	Fehlerklassifikation
eine Transaktion	- Verletzung von Systemrestriktionen • Verstoß gegen Sicherheitsbestimmungen • übermäßige Betriebsmittelanforderungen - anwendungsbedingte Fehler • z. B. falsche Operationen und Werte	Transaktionsfehler
mehrere Transaktionen	- geplante Systemschließung - Schwierigkeiten bei der Betriebsmittelvergabe • Überlast des Systems • Verklemmung mehrerer Transaktionen	
alle Transaktionen (das gesamte Systemverhalten)	- Crash mit Verlust der Hauptspeicherinhalte • Hardware-Fehler • falsche Werte in kritischen Tabellen	Systemfehler (Crash)
	- Zerstörung von Sekundärspeichern	Gerätefehler
	- Zerstörung des Rechenzentrums	Katastrophen

Recovery-Arten

1. Transaktions-Recovery

Zurücksetzen einzelner (noch nicht abgeschlossener) TA
im laufenden DB-Betrieb (TA-Fehler, Deadlock usw.)[5]

- R1: vollständiges Zurücksetzen auf BOT (TA-Undo) bzw.
- R0: partielles Zurücksetzen auf Rücksetzpunkt (*Savepoint*) innerhalb der Transaktion

2. Crash-Recovery nach Systemfehler

Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustandes:

- R2: (partielles) Redo für erfolgreiche TA
  (Wiederholung verlorengegangener Änderungen)
- R3: Undo aller durch Ausfall unterbrochenen TA
  (Entfernen der Änderungen aus der permanenten DB)

3. Medien-Recovery nach Gerätefehler (R4)

- 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 der Basis gesicherter Archivkopien (Datenverlust!)

---

5. Die verschiedenen Recovery-Verfahren werden auch mit R0 - R4 abgekürzt.

Recovery-Arten (2)

- **A Fundamental Theorem of Recovery**

  Axiom 1 (Murphy): All programs (DBMSs) are buggy.

  Theorem 1 (Law of Large Programs):
  Large programs are even buggier than their size would indicate.

  Corollary 1.1:
  A recovery-relevant program has recovery bugs.

  Theorem 2:
  If you do not run a program, it does not matter
  whether or not it is buggy.

  Corollary 2.1:
  If you do not run a program, it does not matter
  whether or not it has recovery bugs.

  Theorem 3:
  Exposed machines should run as few programs as possible;
  the ones that are run should be as small as possible!???

  ➥ ***KISS: Keep It Simple, Stupid!***

- **Annahmen**

  (Unter welchen Voraussetzungen funktioniert die Wiederherstellung der Daten?)

  - quasi-stabiler Speicher
  - fehlerfreier DBMS-Code
  - fehlerfreie Log-Daten
  - Unabhängigkeit der Fehler

Recovery-Arten (3)

- **Pessimistische Variante von „Murphy's Law"**

  ➥ **Was ist zu tun, wenn . . .?**

- **Nicht systematisierte Recovery-Verfahren**

  - R5-Recovery

    - Log-Daten sind fehlerhaft oder DB-Strukturen (ohne Log-Daten) sind unbrauchbar
    - kein TA-konsistenter, bestenfalls aktions- oder gerätekonsistenter Zustand erreichbar

    ➥ ***Salvation Programs, Scavenger***

  - R6-Recovery:
    Zusammenfassung aller Maßnahmen außerhalb des Systems

    - Kompensations-TA und
    - Behandlung der Auswirkungen (manuell)

- **Entwicklungsziele**

  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 Lecture**)

  - Verfügbarkeit heute (optimistisch):[6]

    - für Web-Sites: 99%
    - für gut administrierte Systeme: 99,99%, höchstens 99,999%
    - zSeries-Plattform („zero downtime" von IBM): 5 9',
      allerdings nicht für das Gesamtsystem

  - Künftige Verfügbarkeit

    - bis 2010: weitere 5 9' (nicht zu erreichen???)
    - . . .

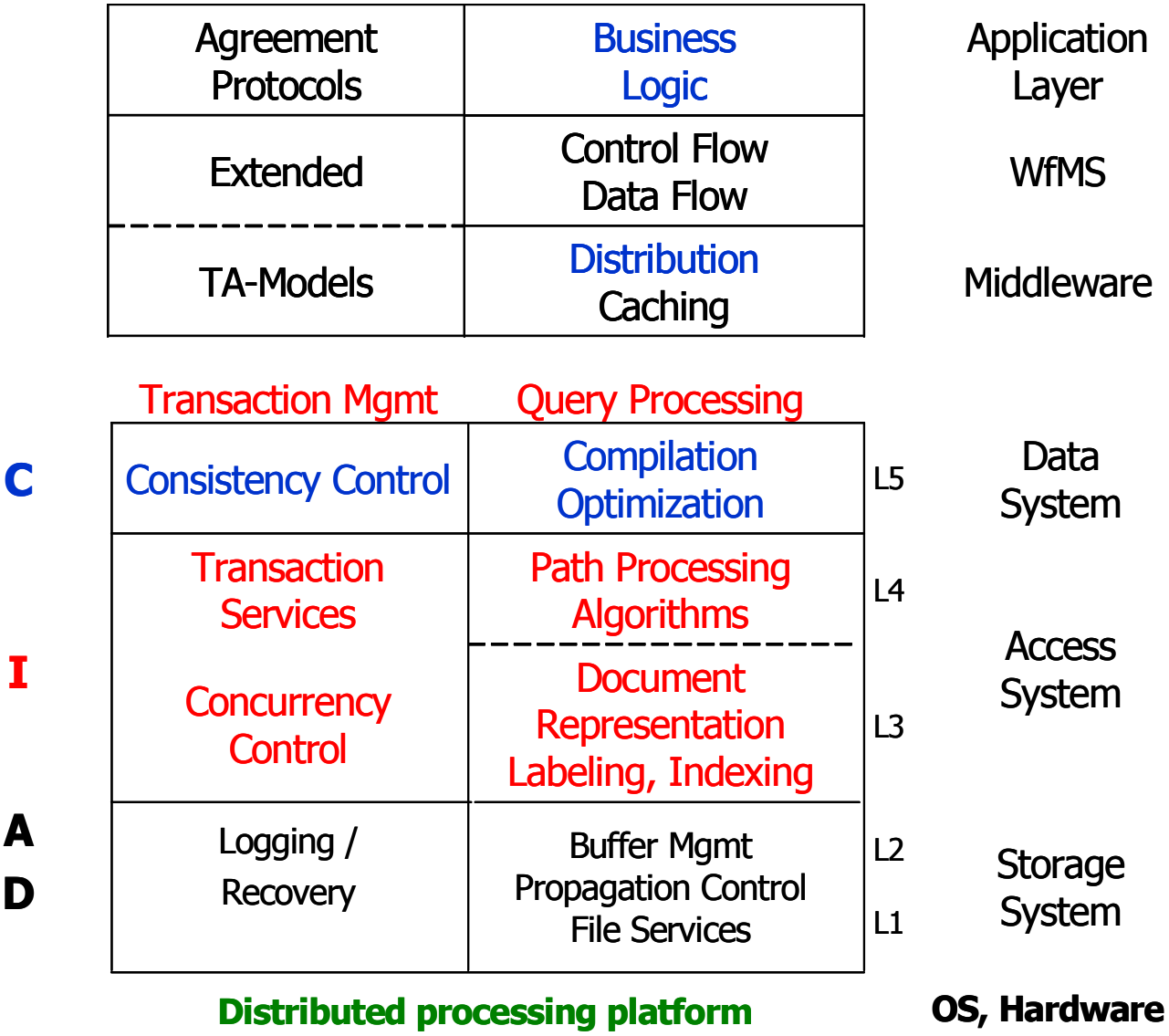
6. Despite marketing campaigns promising 99,999% availability, well-managed servers today achieve 99,9% to 99%, or 8 to 80 hours downtime per year (Armando Fox)

DAIS –The Layer Stack

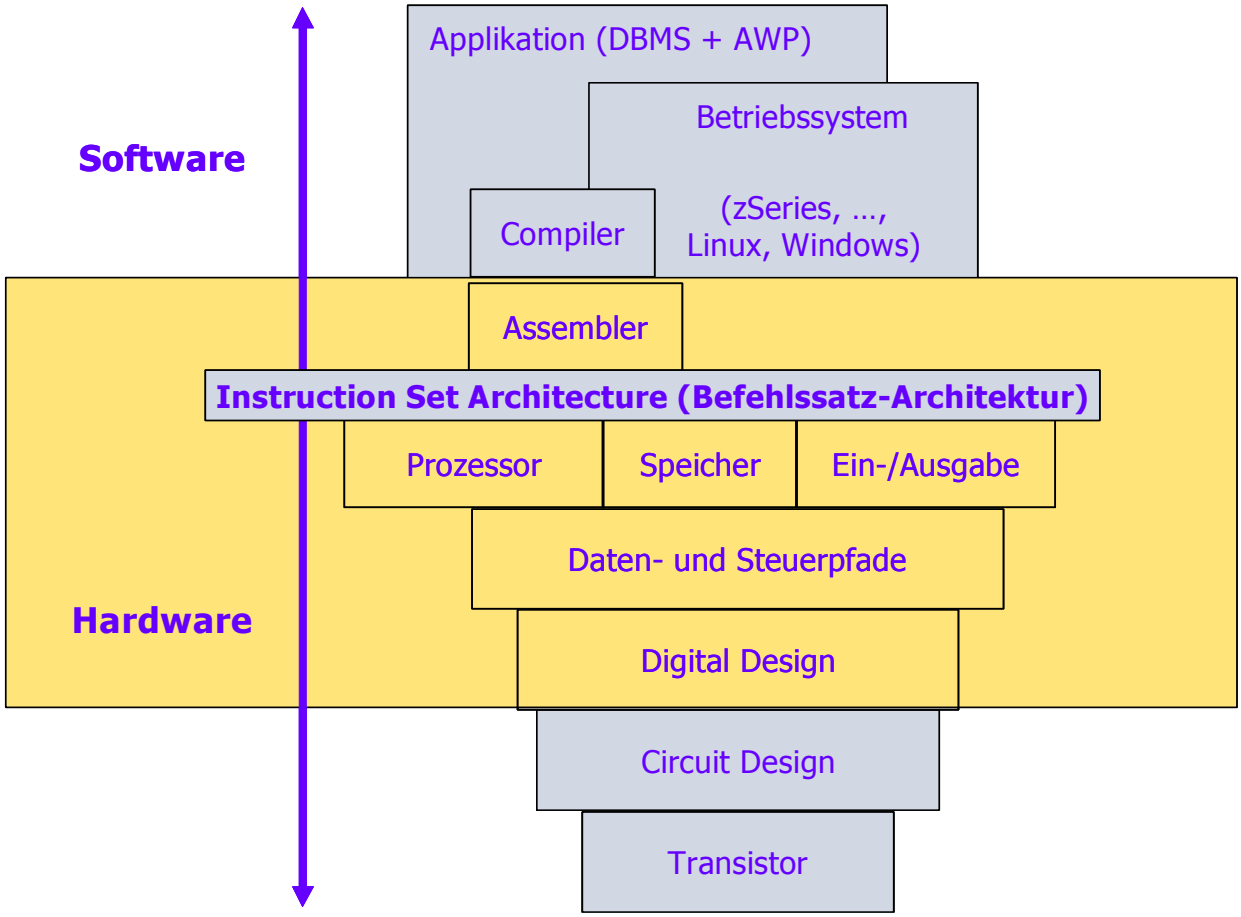
Layer hierarchy in
Dependable Adaptive Information Systems:

DBMS is (only) an important component

Dependability
(availability, reliability, security, safety, confidentiality, integrity, maintainability)
is influenced and determined by all layers!

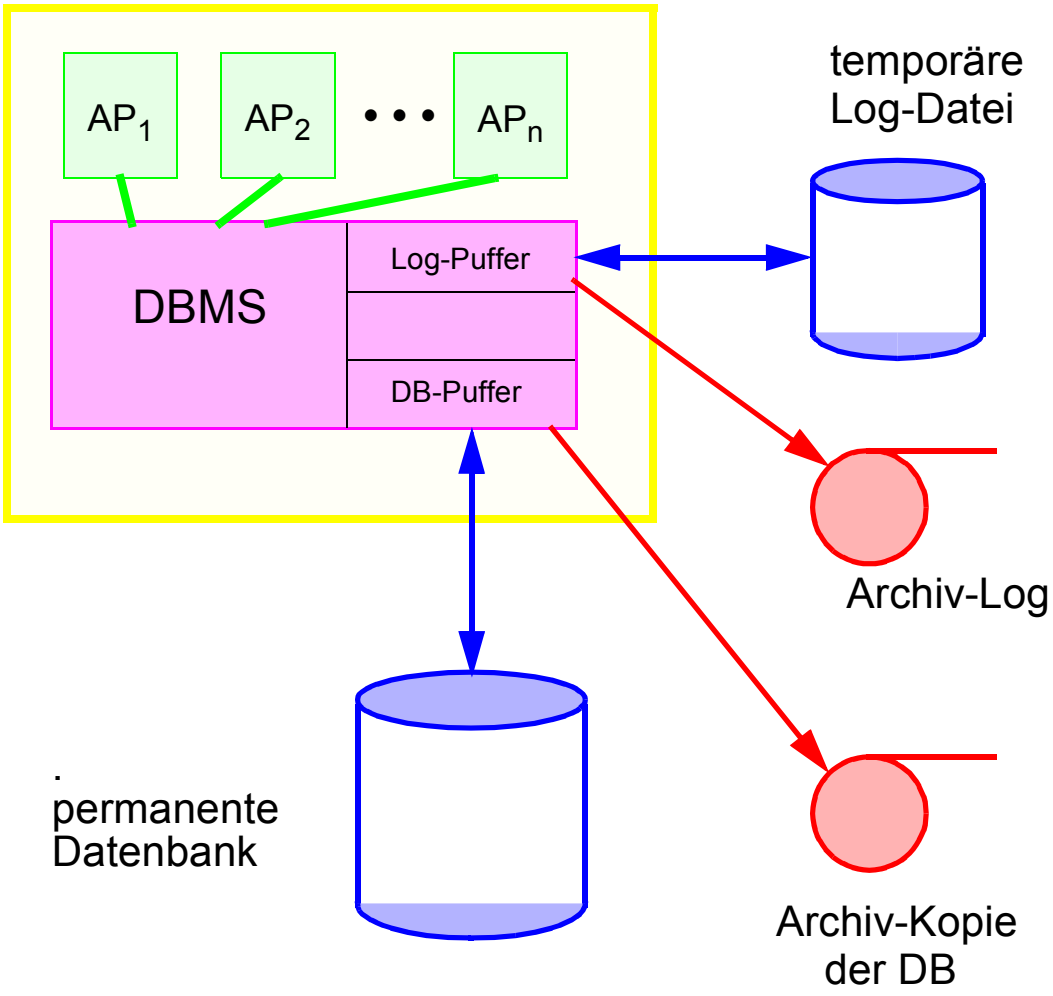


Verarbeitungsplattformen – Virtualisierung aller Ressourcen



- Mainframe-Konzept für heterogene Unternehmens-IT
  - für beherrschbares und koordiniertes Management vieler Ressourcen,
  - Basis hocheffizienter Konsolidierung und Integration verteilter Ressourcen,
  - garantiert verfügbare Infrastruktur für unternehmenskritische Daten und Prozesse.
- Das Fundament, die zSeries-Hardware- und -Firmware-Basis:
  - Performance & Skalierbare Kapazität aller Ressourcen
    - Prozessoren, Datenfluss, Speicher, I/O-Bandbreiten, I/O-Connectivity, Sysplex
  - Extreme Verfügbarkeit der gesamten Infrastruktur
    - Prozessoren, Books, Firmware, I/O, System
  - Flexibilität des Betriebs
    - Dynamische Optimierung aller Systemressourcen in der Architektur verankert
    - Logische und physische Anpassung der Kapazität im laufenden Betrieb
  - Sicherheit des Betriebs
    - Zertifizierte LPAR-Sicherheit, hochsichere kryptographische Hardware

DB-Recovery – Systemkomponenten



- **Pufferung von Log-Daten im Hauptspeicher** (Log-Puffer)
  Ausschreiben spätestens bei Commit
- **Einsatz der Log-Daten**

1. **Temporäre Log-Datei**

   zur Behandlung von Transaktions- und Systemfehlern

   DB + temp. Log ⇒ DB

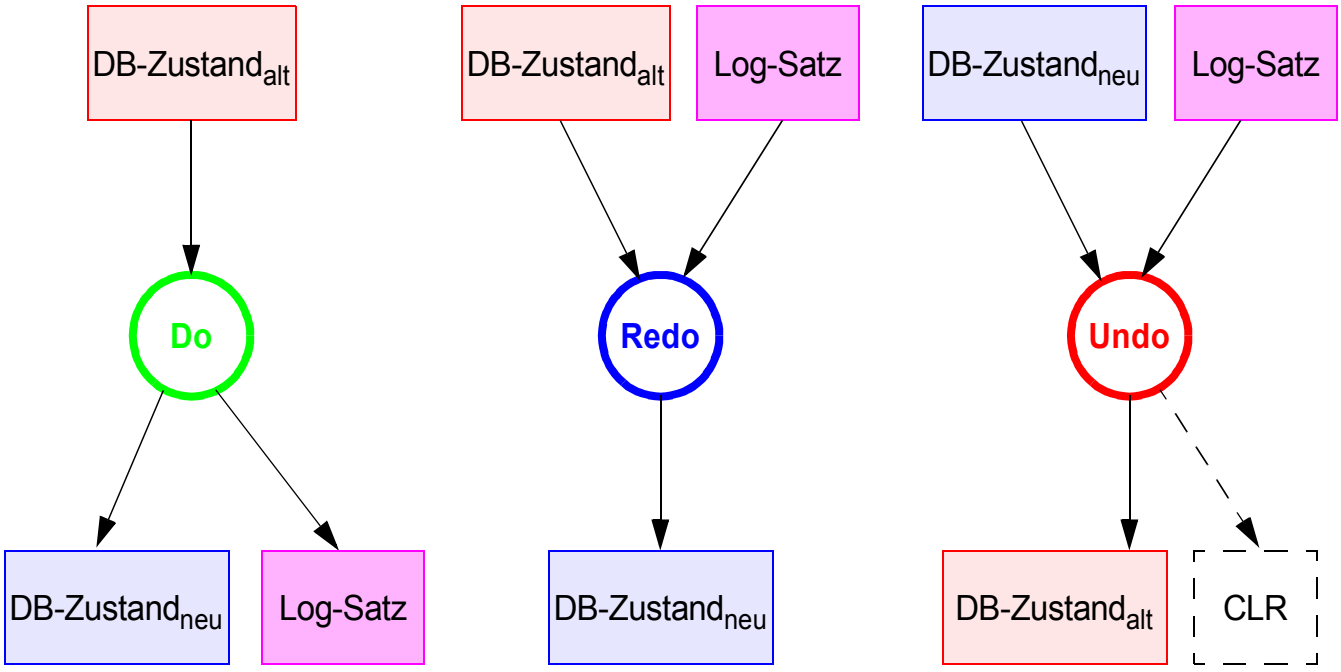
2. **Behandlung von Gerätefehlern:**

   Archiv-Kopie + Archiv-Log ⇒ DB

Logging-Aufgaben

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

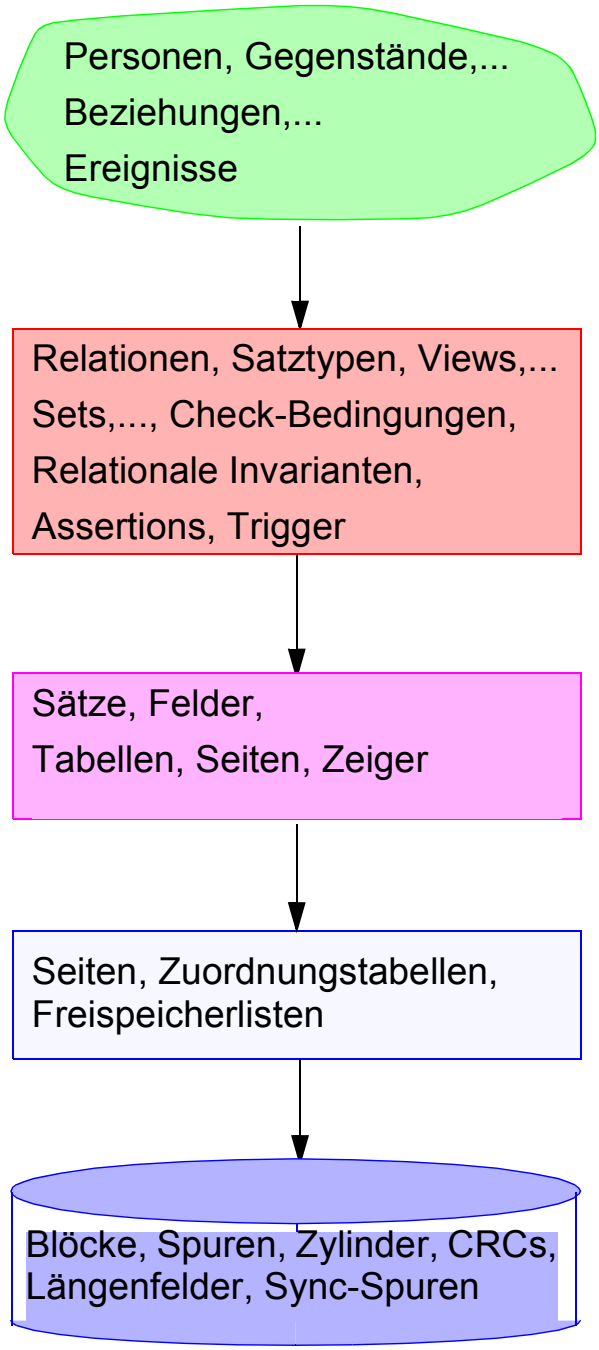
- **Do-Redo-Undo-Prinzip**



CLR = Compensation Log Record (für Crash während der Recovery)

- **Log-Granulat**
  - Welche Granulate können gewählt werden?
  - Was ist zu beachten?

Abstraktionsebenen und Logging



Anwendungsbereich („Miniwelt“) des Benutzers, Semantische Integritätsbedingungen

Datenmodell zur formalisierten Abbildung der Miniwelt, Modellbedingungen (Abb. der semantischen Integritätsbedingungen)

Repräsentation in einem linearen Adressraums, Konsistenz von Tabellen, Zeigern usw.

Repräsentation der Seiten in Dateien, Konsistenz der Verwaltungsdaten usw.

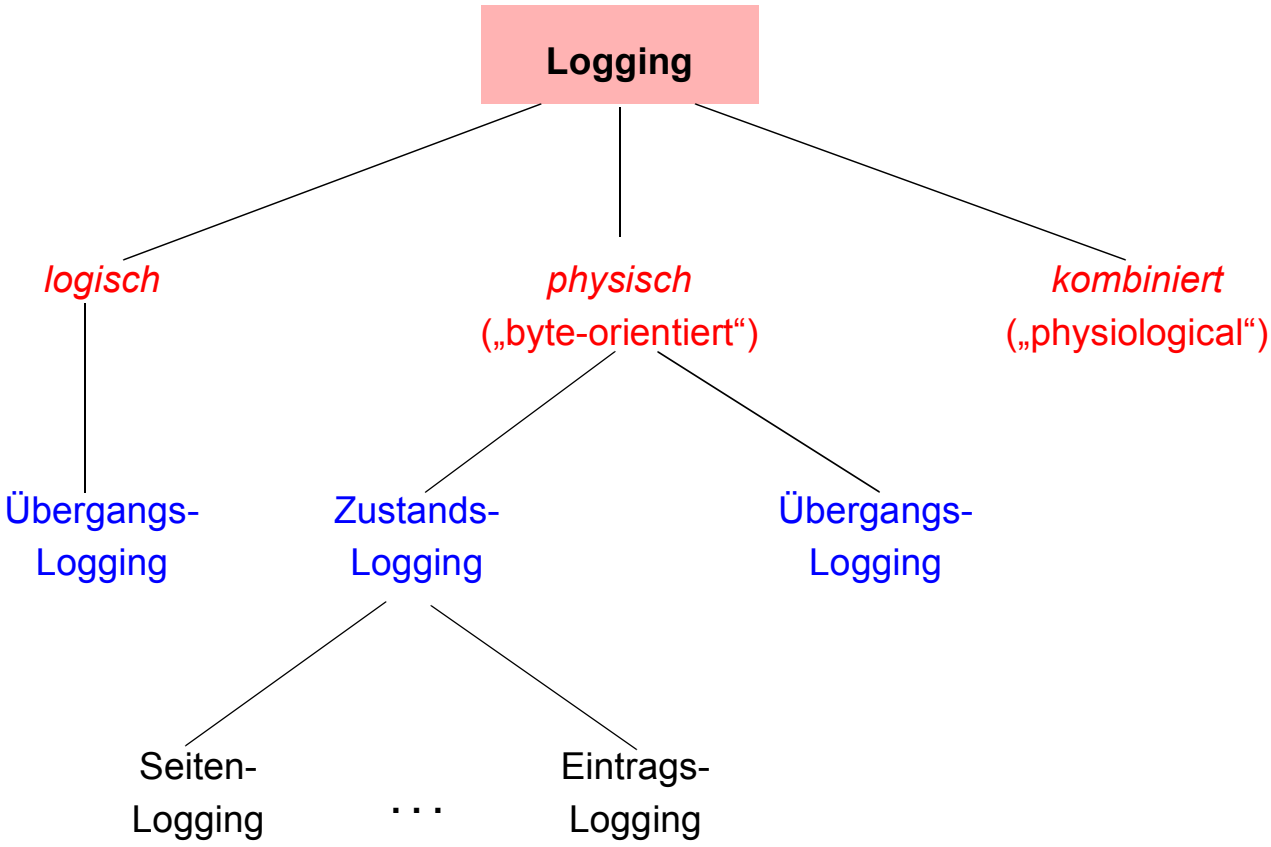
Physische Repräsentation auf externen Speichern, Paritäts-, Längenbedingungen usw.

Logging kann auf jeder Ebene erfolgen:

Das Sammeln von ebenenspezifischer Log-Information setzt voraus, dass **bei Recovery die Konsistenzbedingungen der darunterliegenden Abbildungsschicht** im DB-Zustand erfüllt sind!

➥ Wie kann ebenenspezifische Konsistenz im Fehlerfall garantiert werden?

Klassifikation von Logging-Verfahren



- **Logisches Logging**
  - Protokollierung der ändernden DB-Operationen oder gar der TA-Programme mit ihren Parametern
  - Generelles Problem:
    mengenorientierte Aktualisierungsoperation
    (z. B. DELETE <relation>)
  - Undo-Probleme vor allem bei nicht-relationalen Systemen
    (z. B. Löschen einer Hierarchie von Set-Ausprägungen (ERASE ALL))
  - **Voraussetzung:**
    Nach einem Crash müssen auf der permanenten Datenbank DB-Operationen ausführbar sein, d. h., sie muss wenigstens logisch konsistent sein (Operationskonsistenz, TA-Konsistenz)

    ➥ ***verzögerte (indirekte) Einbringstrategie erforderlich***

Klassifikation von Logging-Verfahren (2)

- **Physisches Logging**
  - Log-Granulat: Seite vs. Eintrag/Satz
  - **Zustands-Logging:**
    Alte Zustände (Before-Images) und neue Zustände (After-Images) geänderter Objekte werden in die Log-Datei geschrieben
  - **Übergangs-Logging:**
    Protokollierung der Differenz zwischen Before- und After-Image
  - Physisches Logging ist bei direkten und verzögerten Einbringstrategien anwendbar

- **Probleme logischer und physischer Logging-Verfahren**
  - **Logisches Logging:**
    für Update-in-Place nicht anwendbar
  - **Physisches, „byte-orientiertes“ Logging:**
    aufwendig und unnötig starr v.a. bezüglich Lösch- und Einfügeoperationen

- **Synthese: Physiologisches Logging**
  Kombination physische/logische Protokollierung:
  Physical-to-a-page, Logical-within-a-page
  - Protokollierung von **elementaren Operationen innerhalb einer Seite**
  - Jeder Log-Satz bezieht sich auf eine Seite
  - Technik ist mit Update-in-Place verträglich

Logging: Anwendungsbeispiel

- **Änderungen** bezüglich einer Seite A:
  1. Ein Objekt a wird in Seite A eingefügt
  2. In A wird ein bestehendes Objekt b_{alt} nach b_{neu} geändert

- **Zustandsübergänge** von A: $A_1 \xrightarrow{1.} A_2 \xrightarrow{2.} A_3$

	logisch	*physisch*
Zustände		Protokollierung der Before- und After-Images 1. A_1 und A_2 2. A_2 und A_3
Übergänge	Protokollierung der Operationen mit Parameter 1. Insert (a) 2. Update (b_{alt},b_{neu})	Differenzen-Logging 1. $A_1 \oplus A_2$ 2. $A_2 \oplus A_3$

- **Rekonstruktion von Seiten** beim Differenzen-Logging:

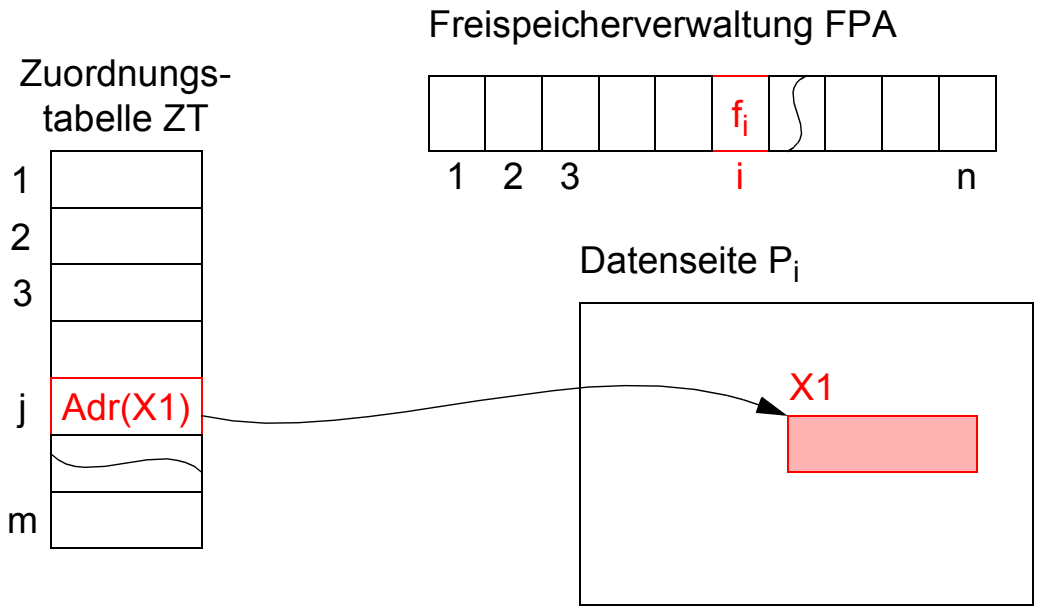
  A_1 als Anfangs- oder A_3 als Endzustand seien verfügbar

  Es gilt:

$A_1 \oplus (A_1 \oplus A_2) = A_2$	$A_3 \oplus (A_2 \oplus A_3) = A_2$
$A_2 \oplus (A_2 \oplus A_3) = A_3$	$A_2 \oplus (A_1 \oplus A_2) = A_1$
Redo-Recovery	Undo-Recovery

Aufwand bei physischem Zustands-Logging

- **Einfachste Form der Implementierung: Seiten-Logging**



- **Operation:** STORE X-RECORD (X1)

Aufwand	Datenseite	ZT	FPA	n Zugriffspfad-seiten (Min)
normaler Betrieb (DO)	neues P_i	Adr(X1)	f_i	n DS_{neu}
Undo-Log	altes P_i	alter Inhalt	alter Inhalt	n DS_{alt}
Redo-Log	neues P_i	Adr(X1)	f_i	n DS_{neu}

Bewertung der Logging-Verfahren

Logging-Verfahren	Logging-Aufwand im Normalbetrieb	Restart-Aufwand im Fehlerfall (Crash)
Seitenzustands-Logging		
Seitenübergangs-Logging		
physiologisches Logging		
Aktions-Logging (Einträge)		
DML-Operations-Logging		
Logging von TA-Programm-parametern		

– – sehr hoch + gering
– hoch ++ sehr gering

- **Vorteile von physiologischem Logging / *Aktions-Logging* gegenüber *Seiten-Logging*:**
  - geringerer Platzbedarf
  - weniger Log-E/As
  - erlaubt bessere Pufferung von Log-Daten (Gruppen-Commit)
  - unterstützt feine Synchronisationsgranulate (Seiten-Logging ➥ Synchronisation auf Seitenebene)

  ➥ **jedoch:** Recovery ist komplexer als mit Seiten-Logging

DB-Konsistenz und Logging

Log-Granulat

Systemhierarchie + DB-Konsistenz im Fehlerfall

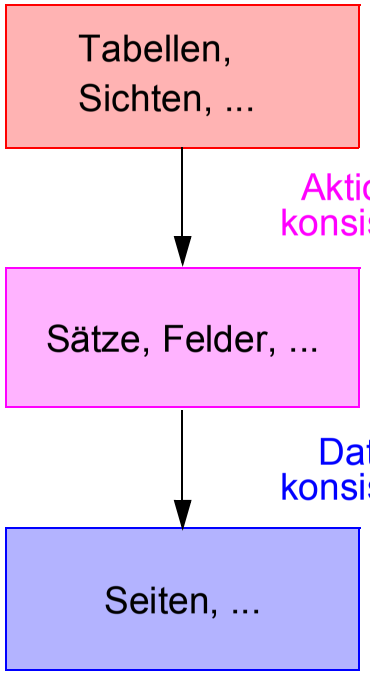
SQL-Operationshierarchie

TA-Programm-parameter

TA-Konsistenz

DML.-Operation

API-Konsistenz



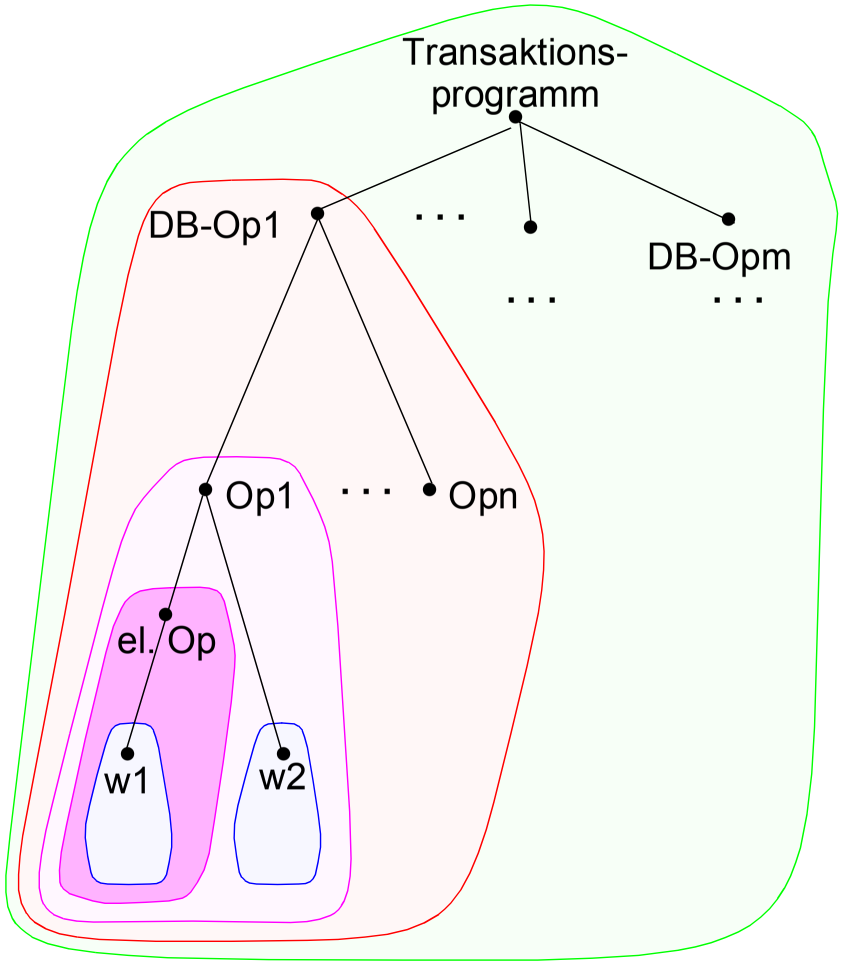
Aktion (Eintrag)

elementare Aktion

Seite

Archiv-Datei/ Archiv-Log

zerstörtes Gerät



DB-Konsistenz und Logging (2)

- **DB-Konsistenz im Fehlerfall**
  - DB-Zustand bei Crash = materialisierte (permanente) DB zum Zeitpunkt des Crashs
  - Eine bestimmte Konsistenz der materialisierten DB bedeutet, dass
    - die Effekte von Operationen der entsprechenden Abstraktionsebene vollständig in die DB eingebracht sind
    - keine Effekte von unvollständigen die DB erreicht haben
    - die Log-Informationen auf den DB-Zustand angewendet werden können
  - Wenn eine Einbringoperation beim Crash unterbrochen wurde, ist der Block i. Allg. unvollständig geschrieben!

    ➥ **Nur beim Seiten-Logging ruft ein unvollständig geschriebener Block keine Medien-Recovery hervor!**

- **Auswahl eines Logging-Verfahrens**

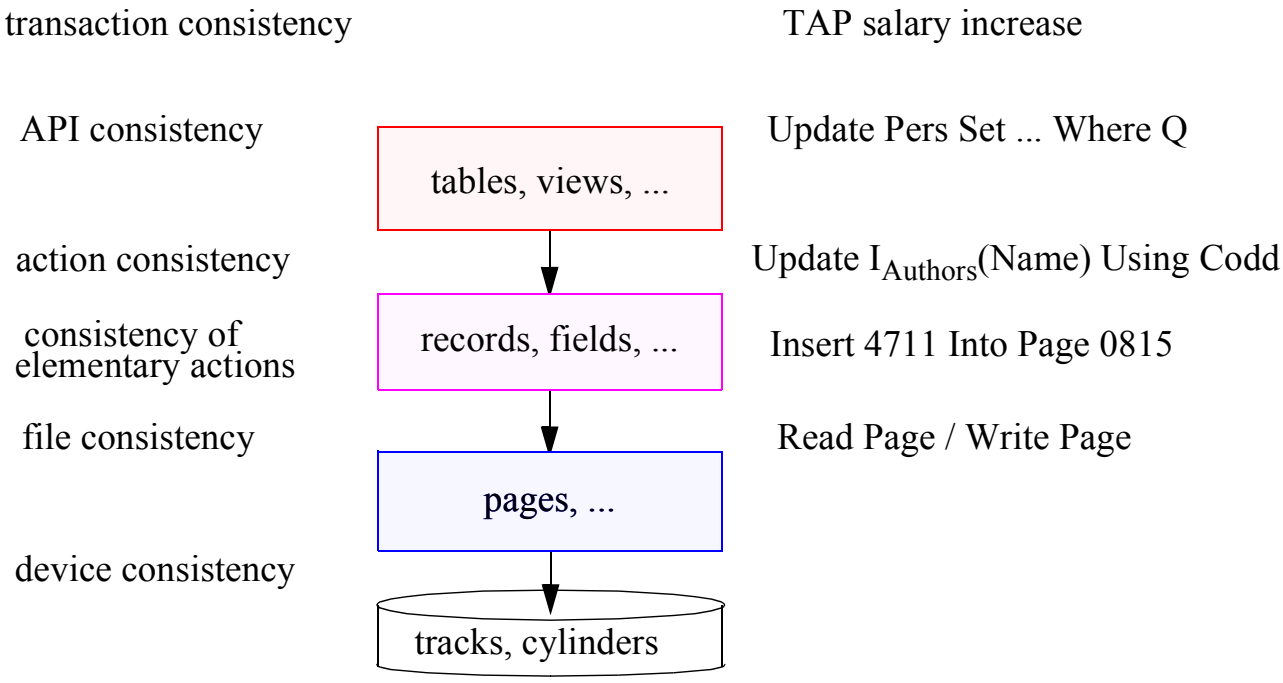
  Wenn im Fehlerfall (Crash) die DB folgende Konsistenz aufweist:
  - Dateikonsistenz → Seiten-Logging (physisch)
  - Konsistenz elementarer Aktionen (EAC) → Physiologisches Logging
  - Aktionskonsistenz (für interne Operationen) → Aktions-Logging (logisch)
  - API-Konsistenz → DML-Op.-Logging (logisch, SQL-Ops)
  - TA-Konsistenz → TA-Programmparameter-Logging (logisch)

    ➥ **Der umgekehrte Schluss ist nicht zwingend!**

- **Garantie einer bestimmten Konsistenz**
  - Wenn bei Crash die Konsistenz einer Abstraktionsebene garantiert wird, können **Logging-Verfahren niedrigerer Konsistenzebene** gewählt werden
  - Dieser Fall tritt üblicherweise nicht auf, da die Gewährleistungskosten für die Konsistenz mit der Abstraktionsebene steigen!

DB-Konsistenz und Logging (2)

- **Zusammenfassung: Systemebenen und Konsistenz**



- **Zusammenfassung: Anforderungen der DB-Konsistenzebenen an die anzuwendenden Log-Verfahren**

consistency level at restart	adjusted log information
file consistency	pages (before- and after-images)
elementary action consistency	physiological logging
action consistency	actions (entries)
API consistency	DML operations
transaction consistency	transaction program invocations with params

Aufbau der (temporären) Log-Datei

- **Verschiedene Satzarten erforderlich**
  - BOT-, Commit-, Abort-Satz
  - Änderungssatz (Undo-Informationen (z. B. ‚Before-Images') und Redo-Informationen (z. B. ‚After-Images'))
  - Sicherungspunktsätze

- **Protokollierung von Änderungsoperationen**
  - **Struktur der Log-Einträge**
    [LSN, TAID, PageID, Redo, Undo, PrevLSN]
  - **LSN** (Log Sequence Number)
    - eindeutige Kennung des Log-Eintrags
    - LSNs müssen monoton aufsteigend vergeben werden
    - chronologische Reihenfolge der Protokolleinträge kann dadurch ermittelt werden
  - **Transaktionskennung** TAID
    der TA, welche die Änderung durchgeführt hat
  - **PageID**
    - Kennung der Seite, auf der die Änderungsoperation vollzogen wurde
    - Wenn die Änderung mehr als eine Seite betrifft, müssen entsprechend viele Log-Einträge generiert werden
  - **Redo**
    Redo-Information gibt an, wie die Änderung nachvollzogen werden kann
  - **Undo**
    Undo-Information beschreibt, wie die Änderung rückgängig gemacht werden kann
  - **PrevLSN**
    ist ein Zeiger auf den vorhergehenden Log-Eintrag der jeweiligen TA. Diesen Eintrag benötigt man aus Effizienzgründen

Beispiel einer Log-Datei

Schritt	T_1	T_2	Log
			[LSN, TAID, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1, T_1, **BOT**, 0]
2.	r(A, a_1)		
3.		**BOT**	[#2, T_2, **BOT**, 0]
4.		r(C, c_2)	
5.	$a_1 := a_1 - 50$		
6.	w(A, a_1)		[#3, T_1, P_A, A-=50, A+=50, #1]
7.		$c_2 := c_2 + 100$	
8.		w(C, c_2)	[#4, T_2, P_C, C+=100, C-=100, #2]
9.	r(B, b_1)		
10.	$b_1 := b_1 + 50$		
11.	w(B, b_1)		[#5, T_1, P_B, B+=50, B-=50, #3]
12.	**Commit**		[#6, T_1, **Commit**, #5]
13.		r(A, a_2)	
14.		$a_2 := a_2 - 100$	
15.		w(A, a_2)	[#7, T2, P_A, A-=100, A+=100, #4]
16.		**Commit**	[#8, T_2, **Commit**, #7]

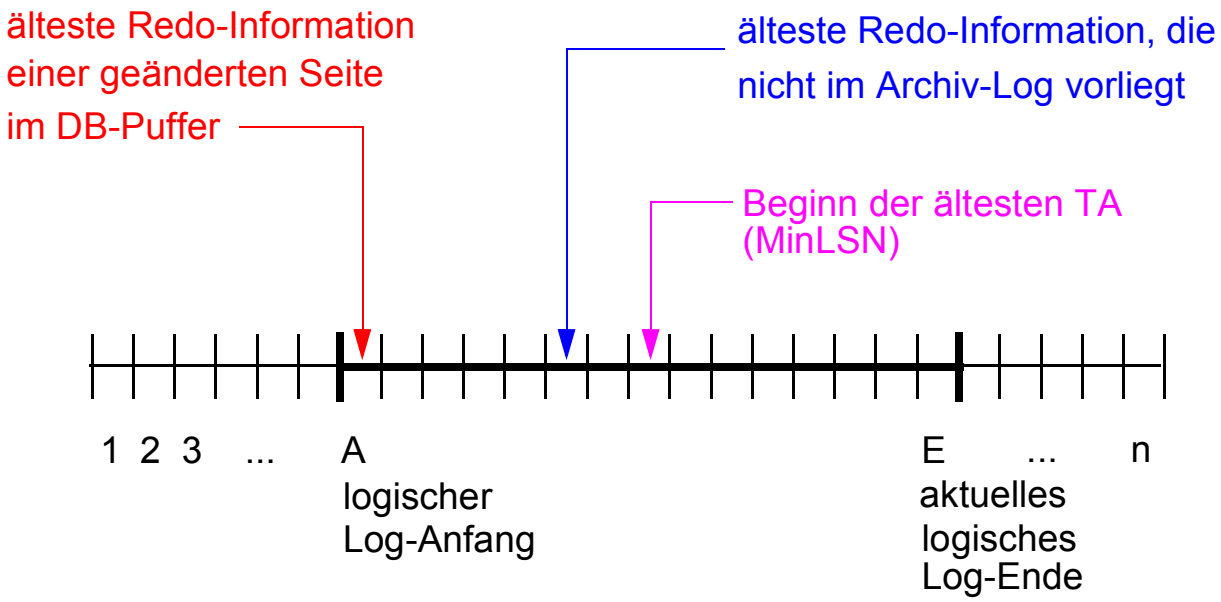
Aufbau der (temporären) Log-Datei (2)

- **Log ist eine sequentielle Datei**

  Schreiben neuer Protokolldaten an das aktuelle Dateiende

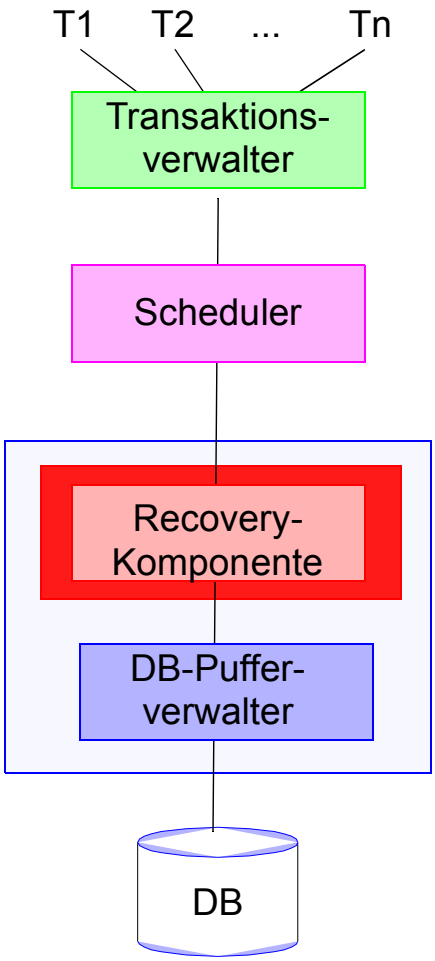
- Log-Daten sind für Crash-Recovery **nur begrenzte Zeit relevant**
  - Undo-Sätze für erfolgreich beendete TA werden nicht mehr benötigt
  - nach Einbringen der Seite in die DB wird Redo-Information nicht mehr benötigt
  - Redo-Information für Medien-Recovery ist im Archiv-Log zu sammeln!

- ***Ringpufferorganisation*** der Log-Datei



Abhängigkeiten zu anderen Systemkomponenten

- **Stark vereinfachtes Modell**



1. **Einbringstrategie für Änderungen**
   - direkt (Non-Atomic, *Update-in-Place*)
   - verzögert (Atomic, Bsp.: Schattenspeicherkonzept)
2. **DB-Pufferverwaltung**
   - Verdrängen ‚schmutziger' Seiten (Steal vs. NoSteal)
   - Ausschreibstrategie für geänderte Seiten (Force vs. NoForce)
3. **Sperrverwaltung**
   (Wahl des Sperrgranulats)

Klassen von Historien

- **Definition: Eine Historie H heißt rücksetzbar, falls immer**
  die schreibende TA (T_j) vor der lesenden TA (T_i) ihr Commit ausführt:

  $c_j <_H c_i$

- **Definition: Eine Historie vermeidet kaskadierendes Rücksetzen,** wenn

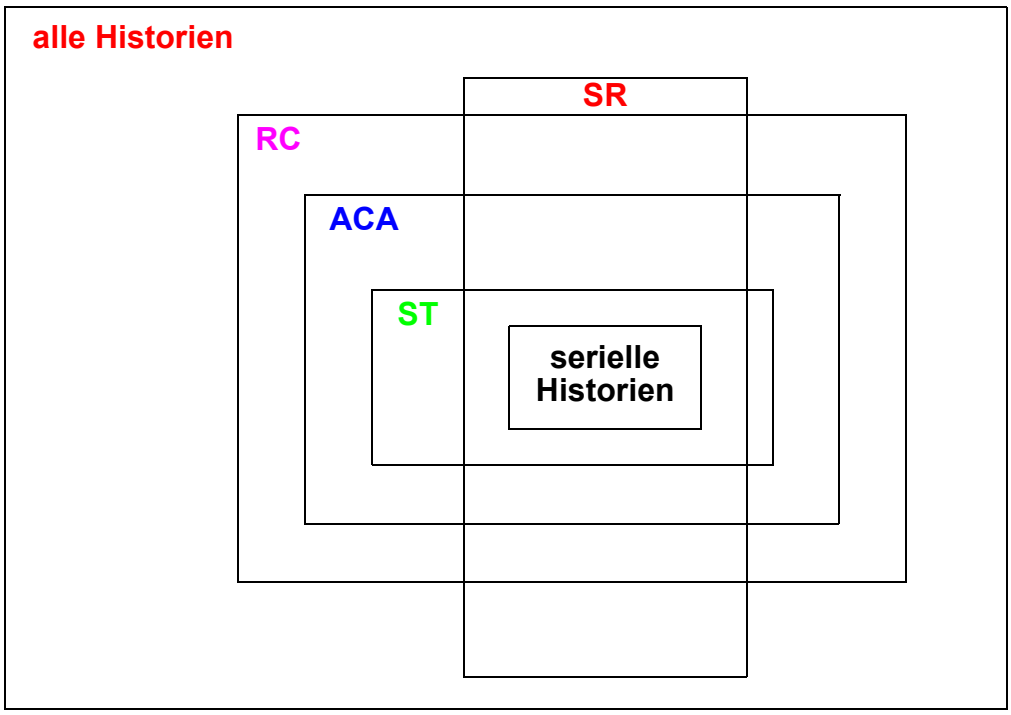
  $c_j <_H r_i[A]$

  gilt, wann immer T_i ein von T_j geändertes Datum liest.

- **Definition: Eine Historie H ist strikt,** wenn für je zwei TA T_i und T_j gilt:

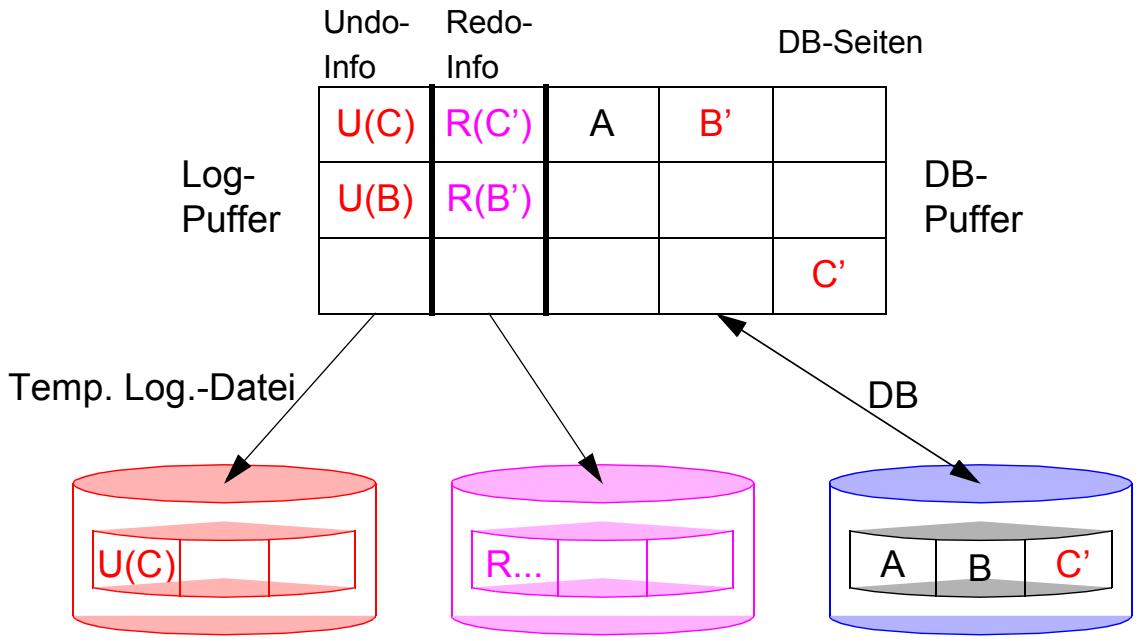
  Wenn
  $w_j[A] <_H o_i[A]$ (mit $o_i = r_i$ oder $o_i = w_i$),
  dann muss gelten:
  $c_j <_H o_i[A]$ oder $a_j <_H o_i[A]$

- **Beziehungen zwischen den Klassen**



Abhängigkeiten zur Einbringstrategie

- **Nicht-atomares Einbringen (Non-Atomic, *Update-in-Place*)**
  - Geänderte Seite wird immer in denselben Block auf Platte zurückgeschrieben
  - Schreiben ist dadurch gleichzeitig Einbringen in die permanente DB (direktes Einbringen)
  - Atomares Einbringen mehrerer geänderter Seiten ist nicht möglich



Undo- und Redo-Info typischerweise in einer sequentiellen Datei

- Es sind **zwei Prinzipien** einzuhalten (Minimalforderung):

1. **WAL-Prinzip: *Write Ahead Log* für Undo-Info**

   U(B) vor B'

2. **Ausschreiben der Redo-Info spätestens bei Commit**

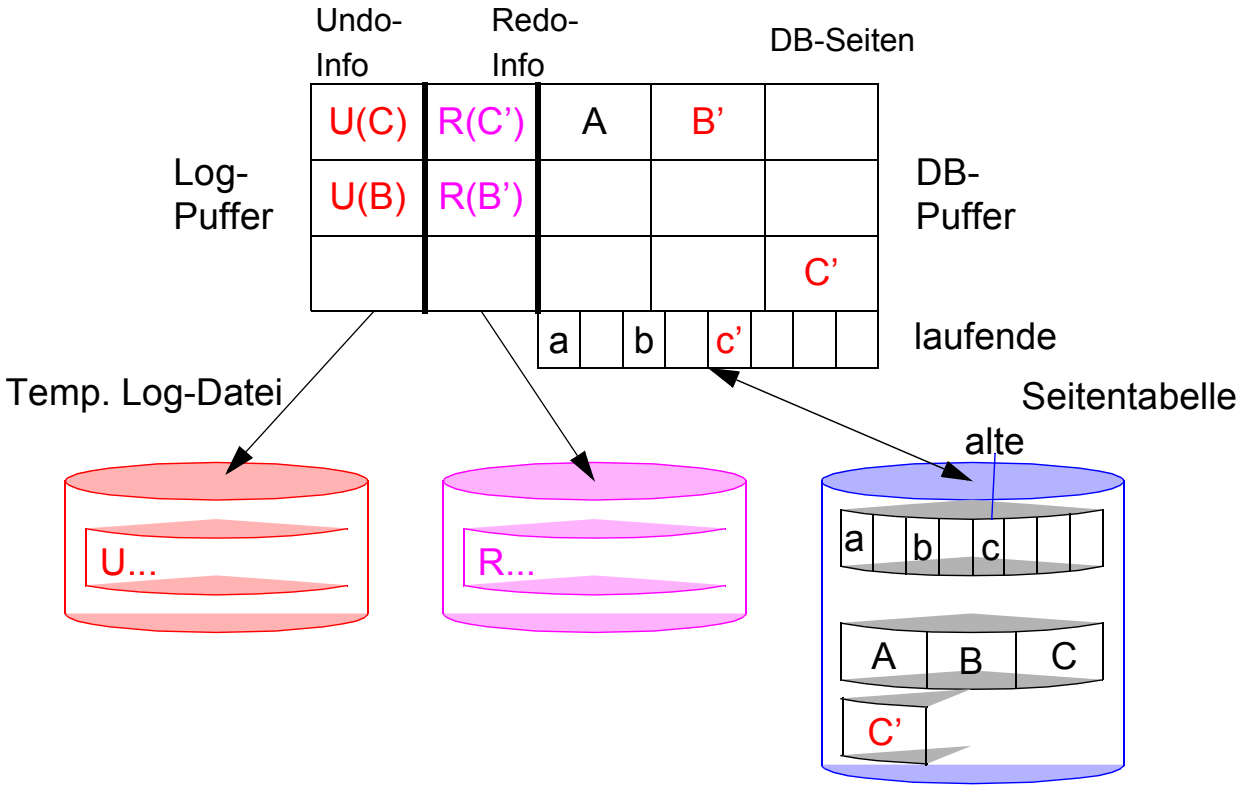
   R(C') + R(B') vor Commit

Abhängigkeiten zur Einbringstrategie (2)

- **Atomares Einbringen (Atomic, z. B. bei** System R, SQL/DS)
  - Geänderte Seite wird in separaten Block auf Platte geschrieben, Einbringen in die DB erfolgt beim nächsten Sicherungspunkt (verzögertes Einbringen)
  - Laufende Seitentabelle gibt aktuelle Adresse einer Seite an
  - Atomares Einbringen mehrerer geänderter Seiten ist durch Umschalten von Seitentabellen möglich

    ➥ *aktions- oder transaktionskonsistente DB auf Externspeicher*

    (logisches Logging anwendbar)



1. WAL-Prinzip bei verzögertem Einbringen

TA-bezogene Undo-Info ist vor Sicherungspunkt zu schreiben

U(C) + U(B) vor Sicherungspunkt

2. Ausschreiben der Redo-Info spätestens bei Commit

R(C') + R(B') vor Commit

Abhängigkeiten zur Ersetzungsstrategie

- **Problem: Ersetzung ‚schmutziger' Seiten**

- **Steal:**

  Geänderte Seiten können jederzeit, insbesondere vor EOT der ändernden TA, ersetzt und in die permanente DB eingebracht werden

  \+ große Flexibilität bei Seitenersetzung

  \+ effektivere Puffernutzung bei langen TA mit vielen Änderungen

  – Undo-Recovery erforderlich bei TA-Abbruch, Systemfehler usw.

  ➥ Steal erfordert Einhaltung des ***Write-Ahead-Log (WAL)-Prinzips:***

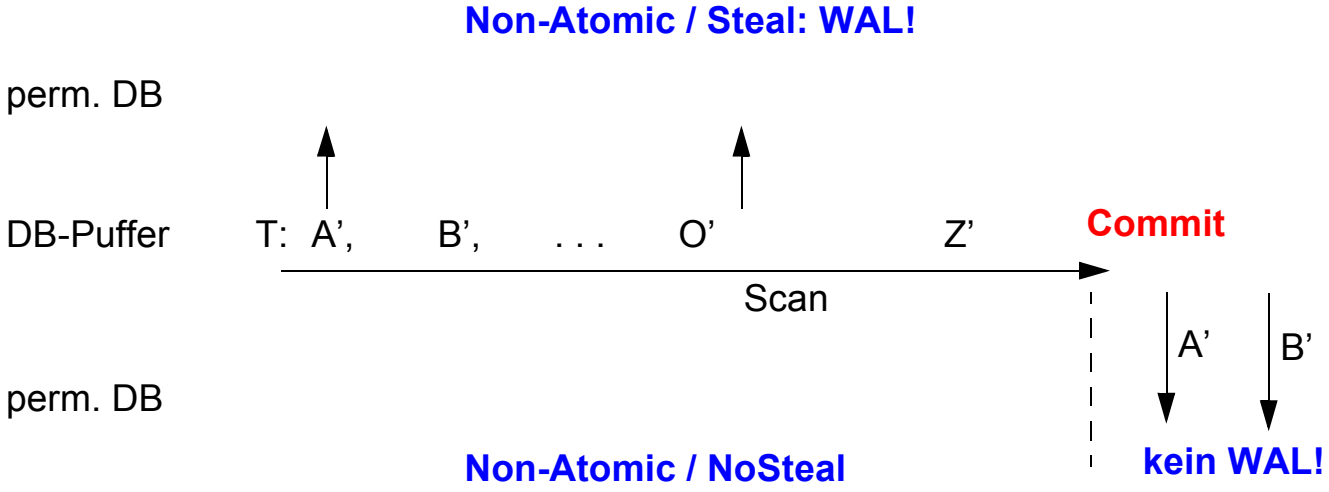
  Vor dem Einbringen einer schmutzigen Änderung müssen zugehörige Undo-Informationen (z. B. Before-Images) in die Log-Datei geschrieben werden

- **NoSteal:**

  \+ keine Undo-Recovery auf der permanenten DB

  – Seiten mit schmutzigen Änderungen dürfen nicht ersetzt werden

- **Beispiel:**



Abhängigkeiten zur Ausschreibstrategie (EOT-Behandlung)

- **Force:**

  Alle geänderten Seiten werden spätestens bei EOT (bei Commit) in die permanente DB eingebracht (Durchschreiben)

  \+ keine Redo-Recovery nach Rechnerausfall

  – hoher Schreibaufwand

  – große DB-Puffer werden schlecht genutzt

  – Antwortzeitverlängerung für Änderungs-TA

- **NoForce:**

  \+ kein Durchschreiben der Änderungen bei EOT

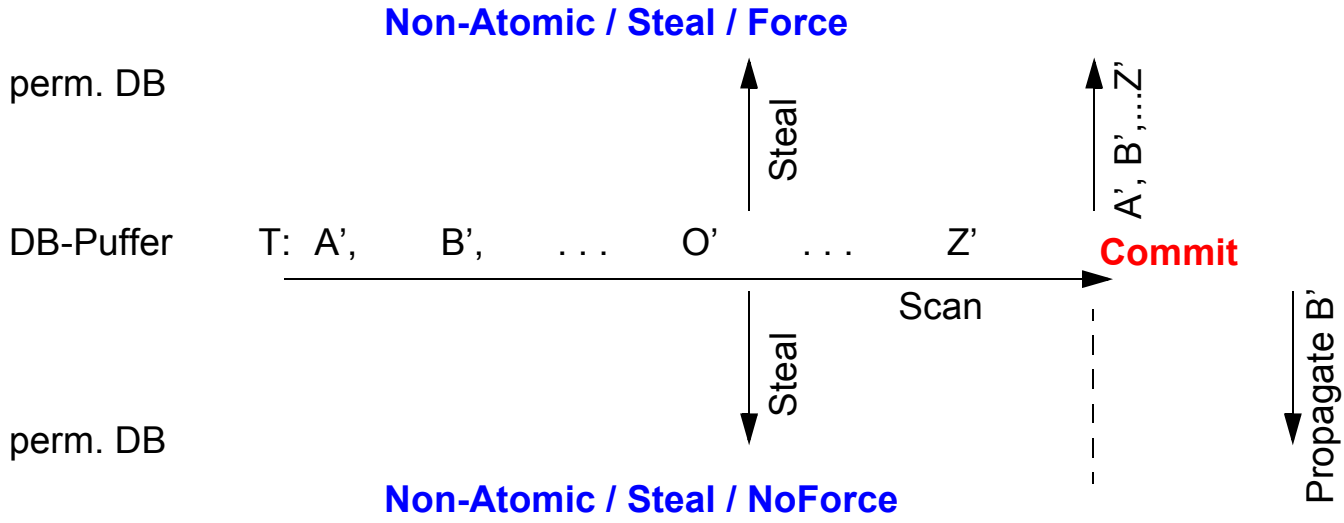
  \+ Beim Commit werden lediglich Redo-Informationen in die Log-Datei geschrieben

  – Redo-Recovery nach Rechnerausfall

- **Commit-Regel:**

  Bevor das Commit einer TA ausgeführt werden kann, sind für ihre Änderungen ausreichende Redo-Informationen (z. B. *After-Images*) zu sichern

- **Beispiel**



Weitere Abhängigkeiten

- **Wie wirken sich Ersetzungs- und Ausschreibstrategie auf die Recovery-Maßnahmen aus?**

	Steal	Nosteal
Force		
Noforce		

- **Abhängigkeit zur Sperrverwaltung**

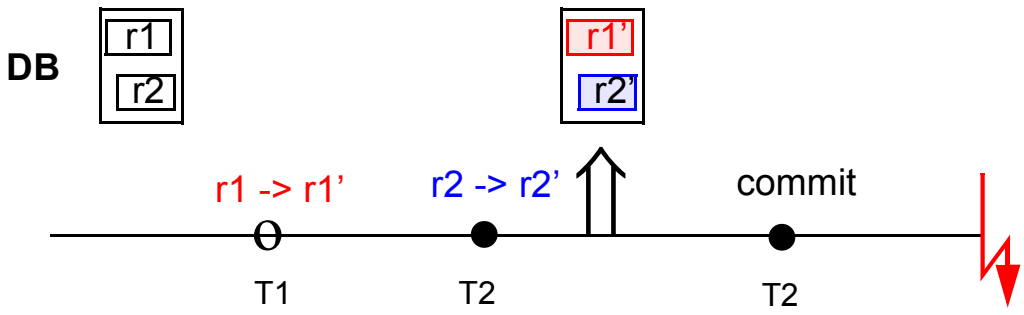
  Log-Granulat muss kleiner oder gleich dem Sperrgranulat sein!

  Beispiel:

  Sperren auf Satzebene,

  Before- bzw. *After-Images* auf Seitenebene

  ➥ Undo (Redo) einer Änderung kann parallel durchgeführte Änderungen derselben Seite überschreiben *(lost update)*



Test zur Fehlerbehandlung

Situation im Fehlerfall (Crash)	Datenseite bereits in die Datenbank eingebracht	Log-Satz bereits in die Log-Datei geschrieben	Transaktion	
			nicht beendet ggf. Zurücksetzung	abgeschlossen ggf. Wiederholung
1.	Nein	Nein		
2.	Nein	Ja		
3.	Ja	Nein		
4.	Ja	Ja		

Mögliche Antworten:

a) Tue überhaupt nichts

b) Benutze die Undo-Information und setze zurück

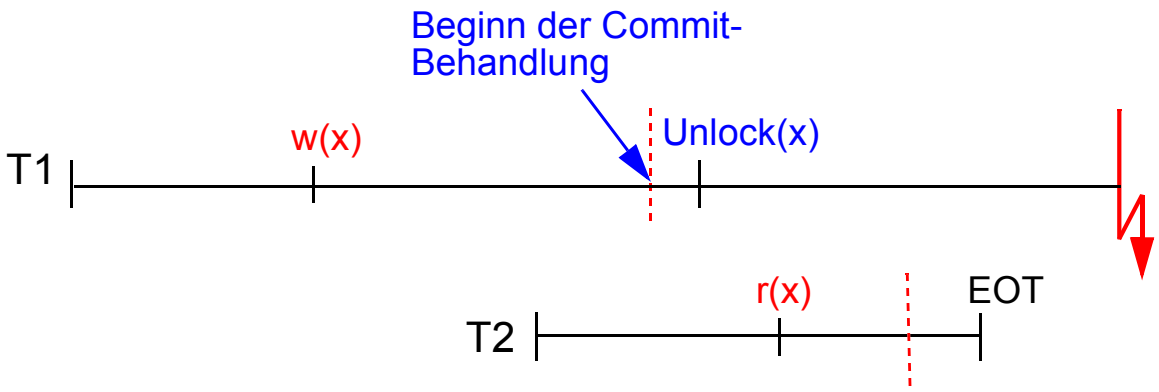
c) Benutze die Redo-Information und wiederhole

d) WAL-Prinzip verhindert diese Situation

e) Zwei-Phasen-Commit-Protokoll verhindert diese Situation

Commit-Behandlung

- **Änderungen einer TA**
  - sind vor Commit zu sichern
  - für andere TA erst sichtbar,
    wenn Durchkommen der ändernden TA gewährleistet ist
  - sonst: Problem des rekursiven Zurücksetzens



- ***Zweiphasige Commit-Bearbeitung***

  Phase 1: Wiederholbarkeit der TA sichern

  - ggf. noch Änderungen sichern
  - Commit-Satz auf Log schreiben

  Phase 2: Änderungen sichtbarmachen (Freigabe der Sperren)

  Benutzer kann nach Phase 1 vom erfolgreichen Ende der TA informiert werden (Ausgabenachricht)

- **Beispiel: Commit-Behandlung bei Force, Steal:**

  1. Before-Images auf Log schreiben
  2. Force der geänderten DB-Seiten
  3. After-Images (für Archiv-Log) und Commit-Satz schreiben

  bei NoForce lediglich 3.) für erste Commit-Phase notwendig

Gruppen-Commit

- **Log-Datei ist potentieller Leistungsengpass**
  - pro Änderungstransaktion wenigstens 1 Log-E/A
  - max. ca. 250 sequentielle Schreibvorgänge pro Sekunde (1 Platte)

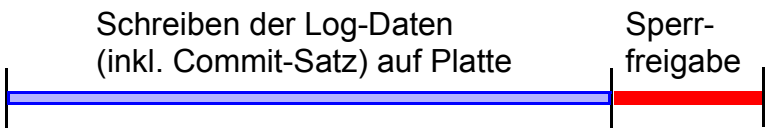
- **Gruppen-Commit:**
  gemeinsames Schreiben der Log-Daten von mehreren TA
  - Pufferung der Log-Daten in Log-Puffer (1 oder mehrere Seiten)
  - Voraussetzung: physiologisches Logging oder Aktions-Logging
  - Ausschreiben des Log-Puffers erfolgt, wenn er voll ist bzw. Timer abläuft
  - nur geringe Commit-Verzögerung

- **Gruppen-Commit**
  erlaubt Reduktion auf 0.1 - 0.2 Log-E/As pro TA
  - Einsparung an CPU-Overhead für E/A reduziert CPU-Wartezeiten
  - dynamische Festsetzung des Timer-Wertes durch DBMS wünschenswert

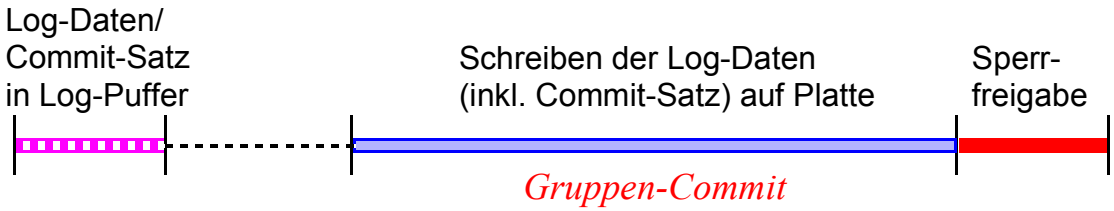
  ➥ Durchsatzverbesserung v.a. bei Log-Engpass oder hoher CPU-Auslastung

Vergleich verschiedener Commit-Verfahren

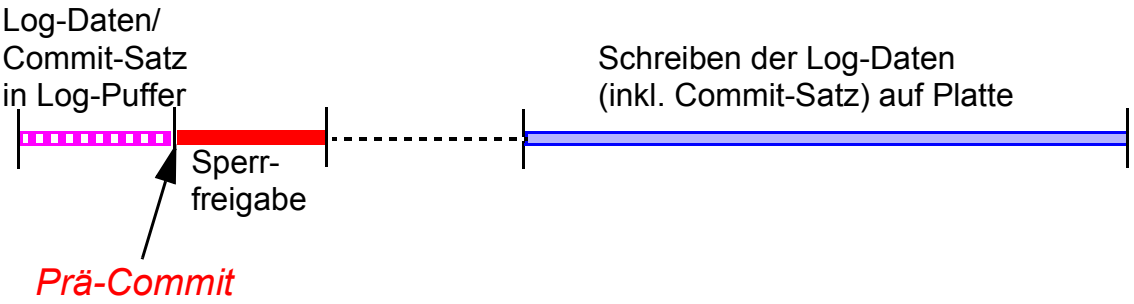
- **Standard-2PC:**



- **Gruppen-Commit:**



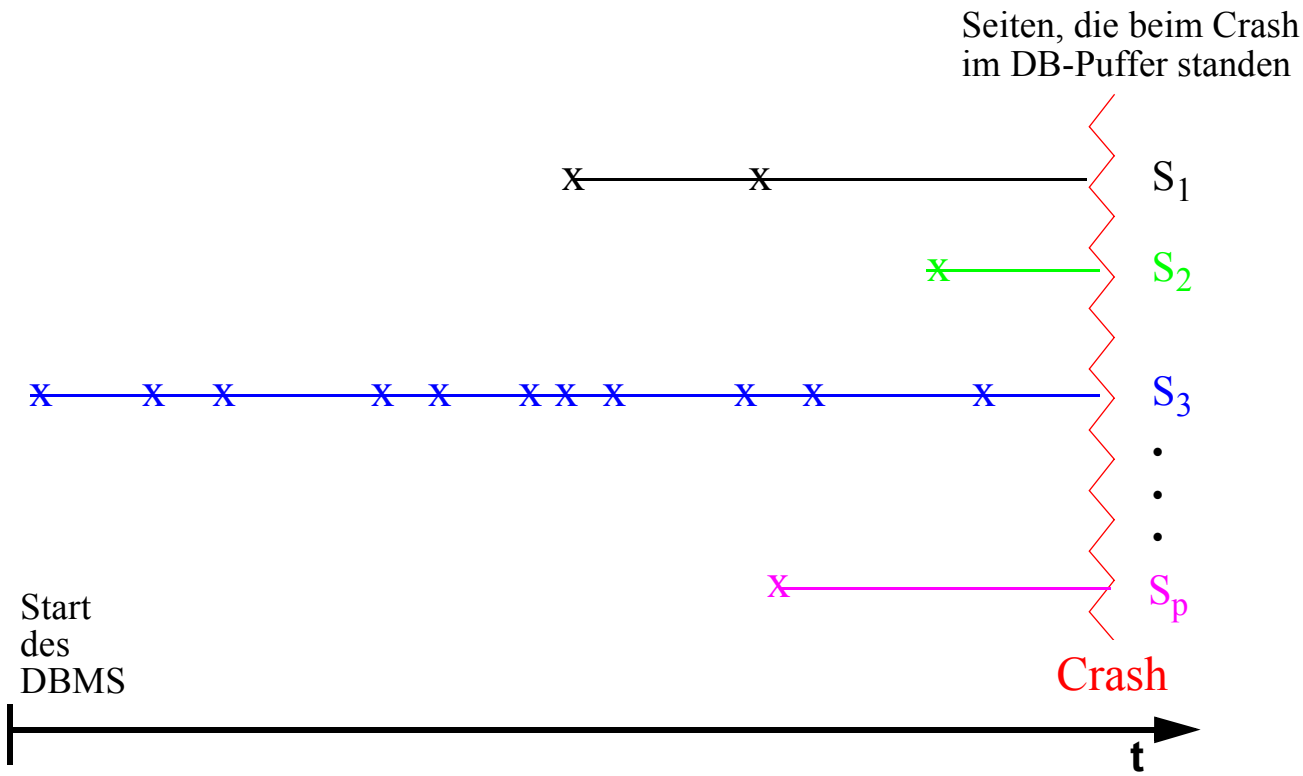
- **Weitere Optimierungsmöglichkeit: *Prä-Commit***
  - Sperren bereits freigeben, wenn Commit-Satz im Log-Puffer steht (vor Schreiben auf Log-Platte)
  - TA kann nur noch durch Systemfehler scheitern
  - In diesem Fall scheitern auch alle ‚abhängigen' TA, die ungesicherte Änderungen aufgrund der vorzeitigen Sperrfreigabe gesehen haben



- In allen drei Verfahren wird der Benutzer erst nach Schreiben des Commit-Satzes auf Platte vom TA-Ende informiert

Sicherungspunkte (*Checkpoints*)

- **Sicherungspunkt**
  - Maßnahme zur Begrenzung des Redo-Aufwandes nach Systemfehlern (Crash)
  - Ohne Sicherungspunkte müssten potentiell alle Änderungen seit Start des DBMS wiederholt werden
  - besonders kritisch: Hot-Spot-Seiten



- **Repräsentation in der Log-Datei**
  - BEGIN_CHKPT-Satz
  - Sicherungspunktinformationen, u. a. Liste der aktiven TA
  - END_CHKPT-Satz
- Log-Adresse des letzten Sicherungspunktsatzes wird in spezieller Restart-Datei geführt

Arten von Sicherungspunkten

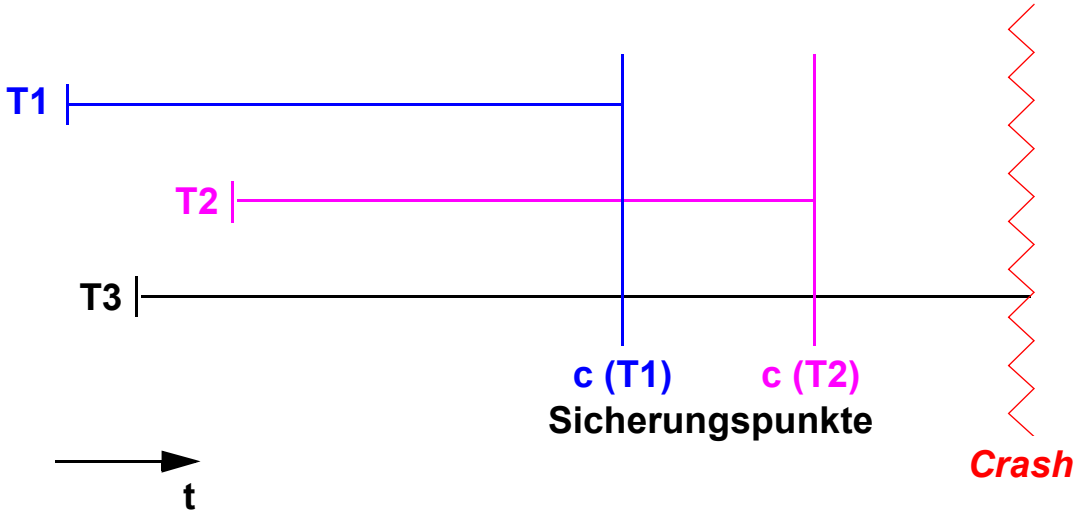
- **Direkte Sicherungspunkte**
  - Alle geänderten Seiten im DB-Puffer werden in die permanente DB eingebracht
  - Redo-Recovery beginnt bei letztem Sicherungspunkt
  - Nachteil: lange „Totzeit“ des Systems, da während des Sicherungspunktes keine Änderungen durchgeführt werden können
  - Problem wird durch große Hauptspeicher verstärkt
  - *Transaktionskonsistente* oder *aktionskonsistente* Sicherungspunkte

- **Indirekte/Unscharfe Sicherungspunkte (Fuzzy Checkpoints)**
  - kein Hinauszwingen geänderter Seiten
  - Nur Statusinformationen (Pufferbelegung, Menge aktiver TA, offene Dateien usw.) werden in die Log-Datei geschrieben
  - sehr geringer Sicherungspunktaufwand
  - Redo-Informationen vor letztem Sicherungspunkt sind i. Allg. noch zu berücksichtigen
  - Sonderbehandlung von Hot-Spot-Seiten

- **Sicherungspunkte und Einbringverfahren**
  - **Atomic**: Zustand der permanenten DB beim Crash entspricht dem zum Zeitpunkt des letzten erfolgreichen Sicherungspunktes
  - **Non-Atomic**: Zustand der permanenten DB enthält alle ausgeschriebenen (eingebrachten) Änderungen bis zum Crash

Transaktionsorientierte Sicherungspunkte

- **Force kann als spezieller Sicherungspunkttyp aufgefasst werden**
  - Nur die Seiten der TA, die Commit durchführt, werden ausgeschrieben
  - Sicherungspunkt bezieht sich immer auf genau eine TA
  - TOC = Transaction-Oriented Checkpoint ≡ Force



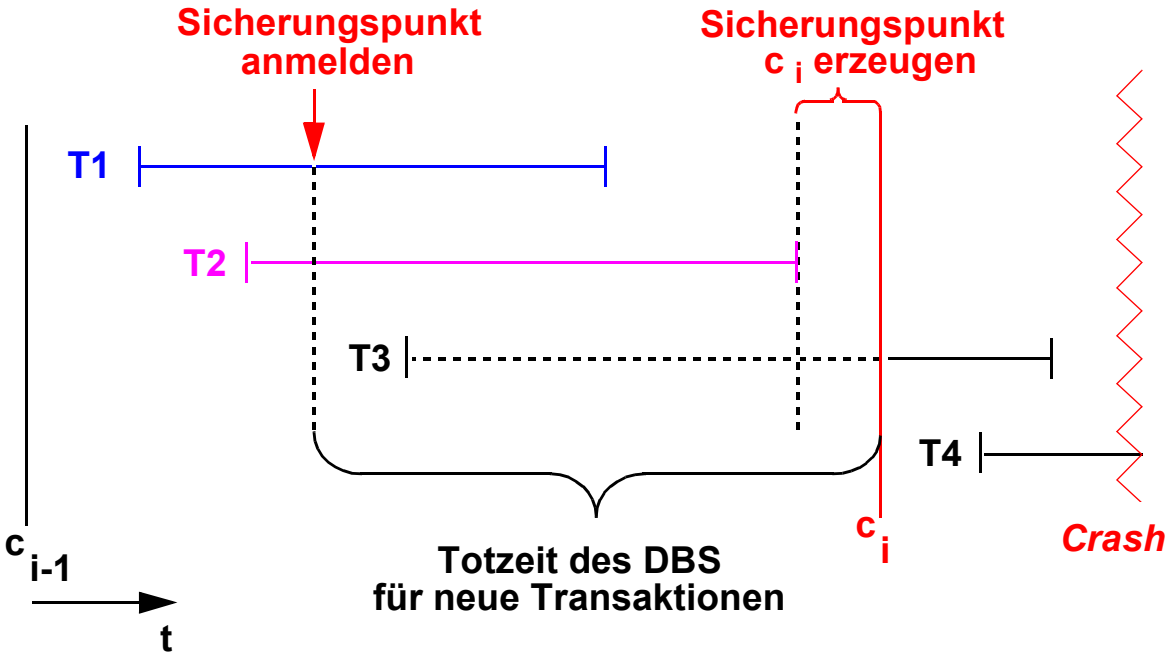
- **Eigenschaften**
  - EOT-Behandlung erzwingt das Ausschreiben aller geänderten Seiten der TA aus dem DB-Puffer
  - bei atomarer Einbringstrategie:
    - DB-Zustand zum Sicherungszeitpunkt bleibt bis zum nächsten erhalten
    - Unter welcher Bedingung bleibt die DB stets **transaktionskonsistent**?

➥ Zumindest bei nicht-atomarem Einbringen der Seiten ist Undo-Recovery vorzusehen (Steal)

- **Abhängigkeit: Non-Atomic, Force => Steal**

Transaktionskonsistente Sicherungspunkte

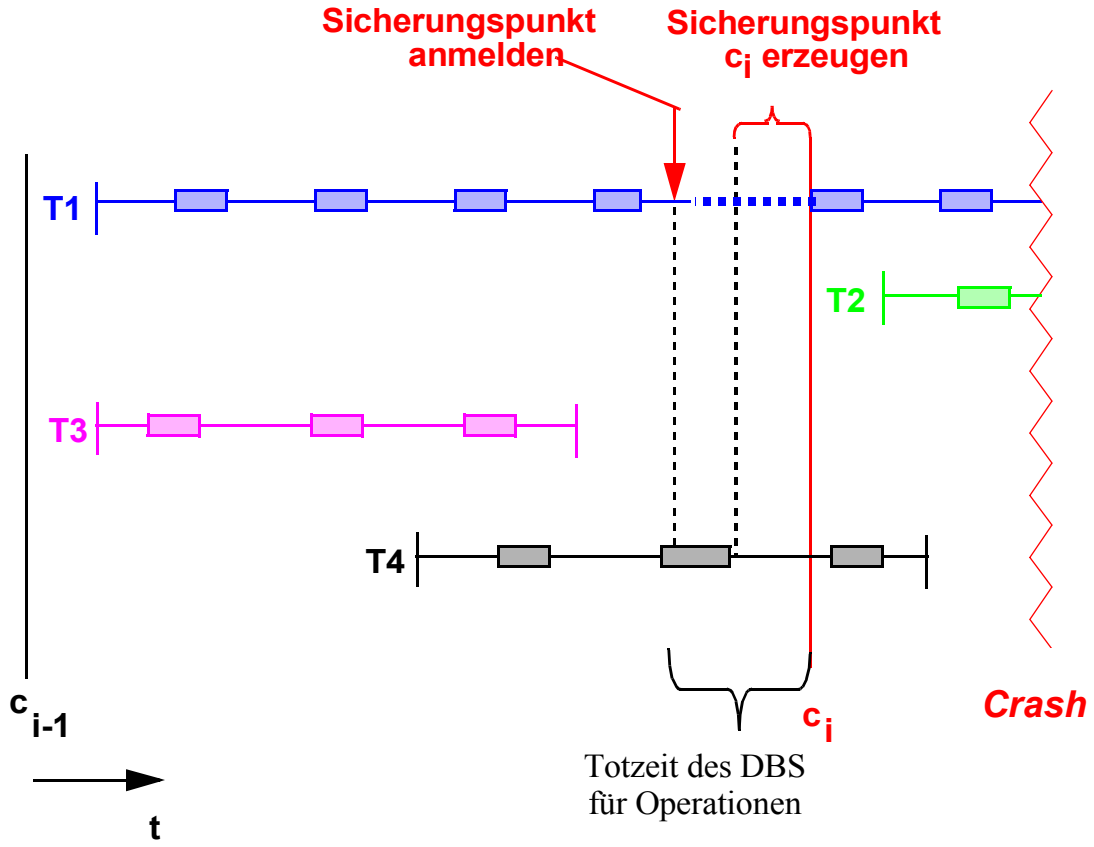
- **Direkter Sicherungspunkt**
  - Sicherungspunkt bezieht sich immer auf alle TA
  - TCC = Transaction-Consistent Checkpoints (logisch konsistent)



- **Eigenschaften**
  - Ausschreiben ist bis zum Ende aller aktiven Änderungs-TA zu verzögern
  - Neue Änderungs-TA müssen warten, bis Erzeugung des Sicherungspunkts beendet ist
  - **Crash-Recovery startet bei letztem Sicherungspunkt (Firewall)**
  - bei atomarer Einbringstrategie:
    - DB-Zustand zum Sicherungszeitpunkt bleibt bis zum nächsten erhalten und
    - ist stets transaktionskonsistent

Aktionskonsistente Sicherungspunkte

- **Direkter Sicherungspunkt**
  - Sicherungspunkt bezieht sich immer auf alle TA
  - ACC = Action Consistent Checkpoints (speicherkonsistent)



- **Eigenschaften**
  - keine Änderungsanweisungen während des Sicherungspunktes
  - geringere Totzeiten als bei TCC,
    dafür Verminderung der Qualität der Sicherungspunkte
  - Crash-Recovery wird **nicht durch letzten Sicherungspunkt begrenzt**
  - bei atomarer Einbringstrategie:
    - DB-Zustand zum Sicherungszeitpunkt bleibt bis zum nächsten erhalten
    - Crash-Recovery vom letzten Sicherungspunkt aus: R2 vorwärts und R3 rückwärts; Reihenfolge von R2- und R3-Recovery ist unerheblich

- **Abhängigkeit: ACC => Steal**

Unscharfe Sicherungspunkte
(Fuzzy Checkpoints)

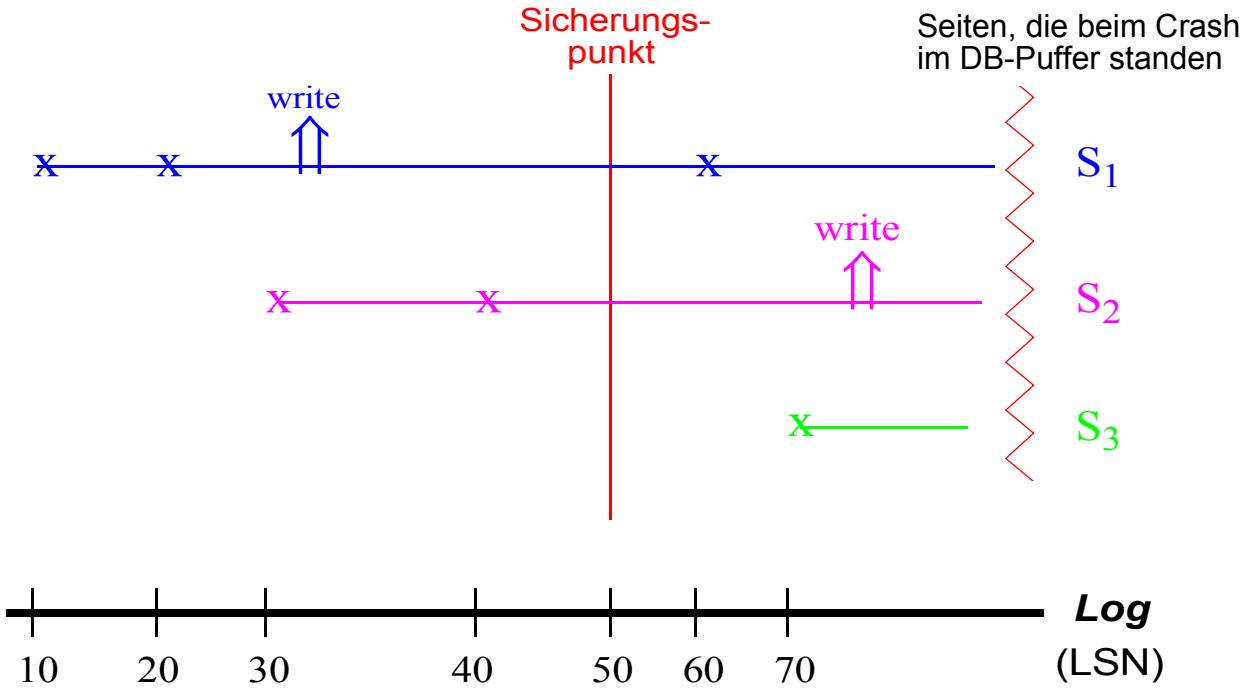
- **DB auf Platte bleibt ‚fuzzy', nicht aktionskonsistent**

  ➥ nur bei Update-in-Place (Non-Atomic) relevant

- **Problem: Bestimmung der Log-Position,** an der **Redo-Recovery** beginnen muss

  - DB-Pufferverwalter vermerkt sich zu jeder geänderten Seite StartLSN, d. h. Log-Satz-Adresse der ersten Änderung seit Einlesen von Platte
  - Redo-Recovery nach Crash beginnt bei MinDirtyPageLSN (= MIN(StartLSN))

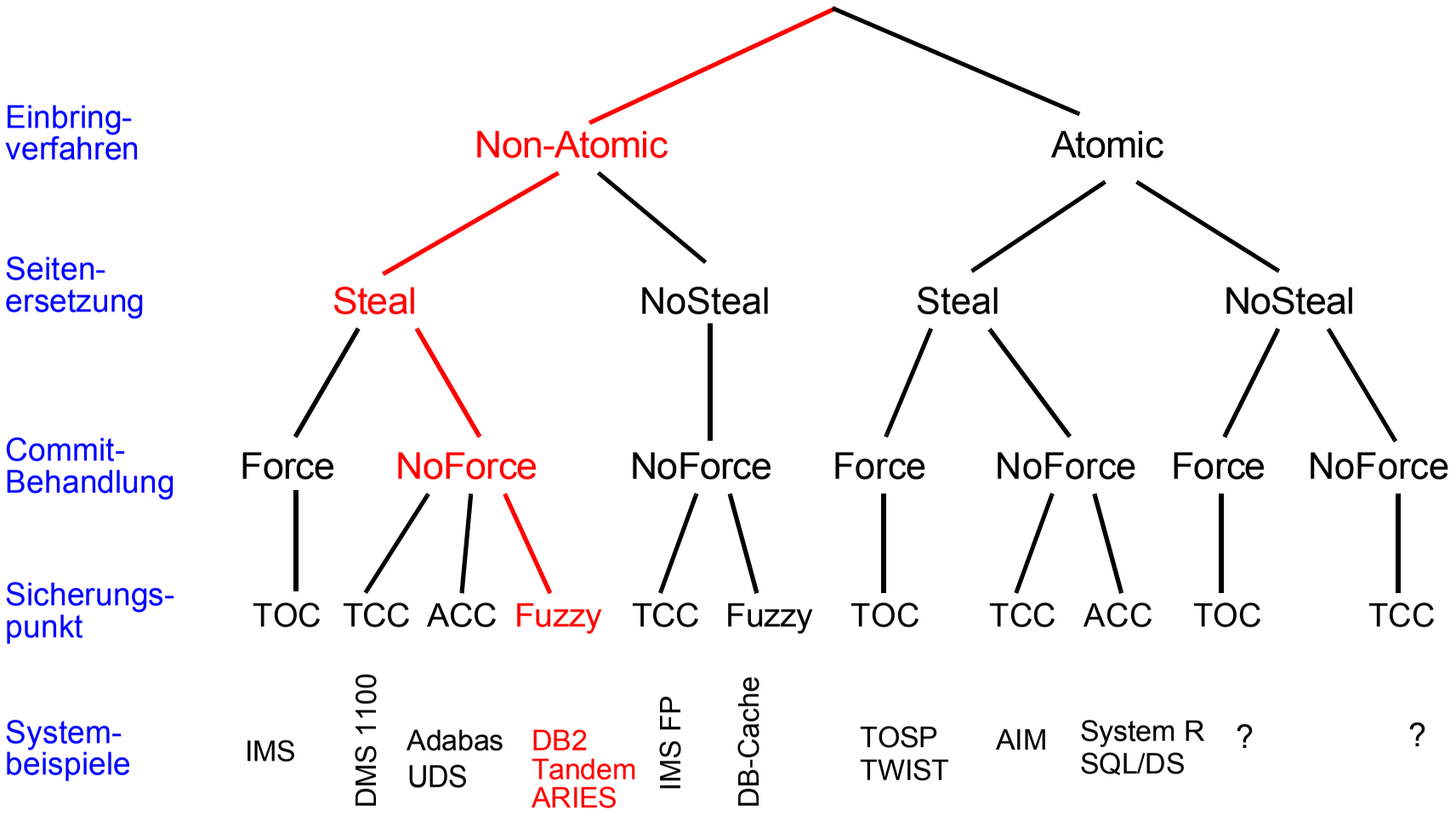
- **Sicherungspunktinformation:**
  MinDirtyPageLSN, Liste der aktiven TA und ihrer StartLSNs, . . .



- **Geänderte Seiten werden asynchron ausgeschrieben**

  - ggf. Kopie der Seite anlegen (für Hot-Spot-Seiten)
  - Seite ausschreiben
  - StartLSN anpassen / zurücksetzen

Klassifikation von DB-Recovery-Verfahren



Nutzung von LSNs

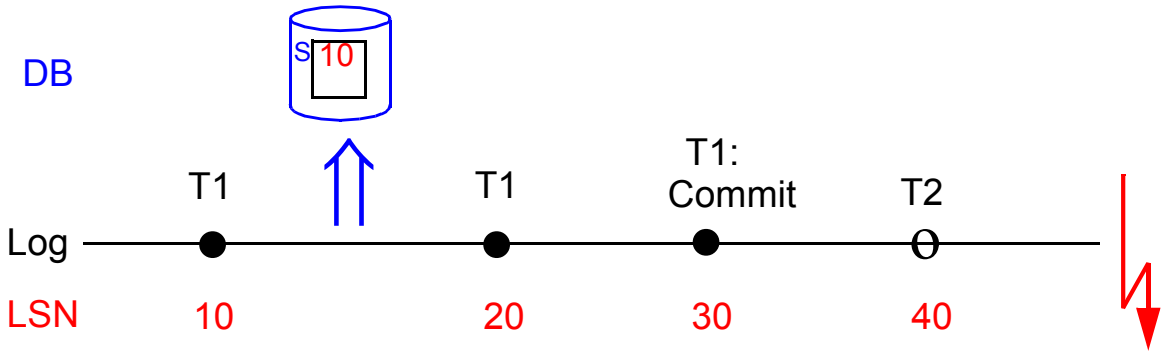
- **Seitenkopf von DB-Seiten enthält Seiten-LSN**
  - Die „Herausforderung“ besteht darin, beim Restart zu entscheiden, ob für die Seite Recovery-Maßnahmen anzuwenden sind oder nicht (ob man den alten oder bereits den geänderten Zustand auf dem Externspeicher vorgefunden hat)
  - Dazu wird auf jeder Seite B die LSN des jüngsten dieser Seite betreffenden Log-Eintrags L gespeichert (PageLSN (B) := LSN (L))

- **Entscheidungsprozedur:**
  Restart hat eine Redo- und eine Undo-Phase
  - **Redo *ist nur erforderlich, wenn***

    Seiten-LSN < LSN des Redo-Log-Satzes
  - **Undo *ist nur erforderlich, wenn***

    Seiten-LSN ≥ LSN des Undo-Log-Satzes

- **Vereinfachte Anwendung: Seitensperren werden vorausgesetzt!**



Redo von T1: S(10) =T1(10) : –

S(10) < T1(20) : Redo, S(20)

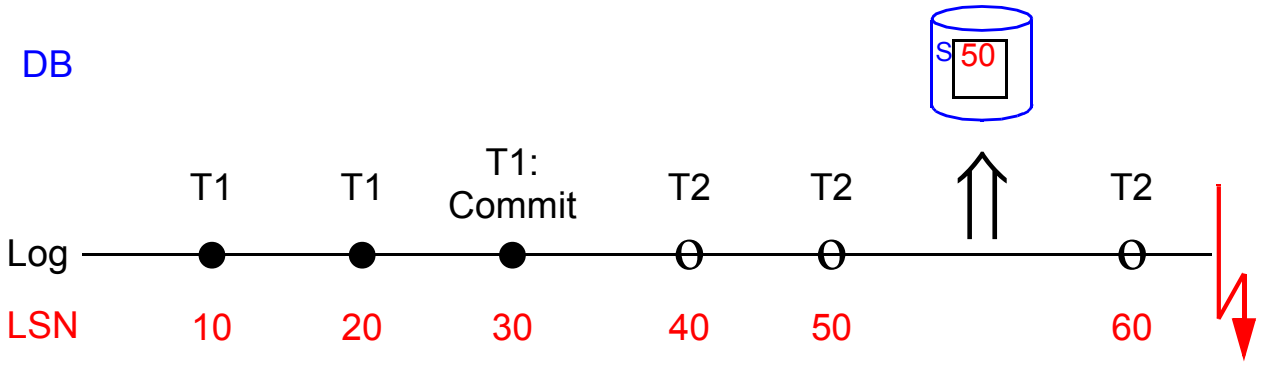
➥ Seiten-LSN wird bei Redo aktualisiert (wächst monoton)

Undo von T2: S(20) < T2(40) : – , **OK!**

➥ Wegen Seitensperren gilt im Recovery-Fall immer:
LSN für Undo (S) > LSN für Redo (S)

Nutzung von LSNs (2)

- **Vereinfachte Anwendung[7]: Seitensperren werden vorausgesetzt!**



Redo von T1: S(50) > T1(10) : –

S(50) > T1(20) : –

Undo von T2: S(50) < T2(60) : –

S(50) ≥ T2(50) : Undo

S(50) ≥ T2(40) : Undo

➥ Was passiert bei Crash im Restart?

- **Undo erfolgt in LIFO-Reihenfolge**
  - Undo muss speziell behandelt werden, so dass wiederholte Ausführung zum gleichen Ergebnis führt (Idempotenz)
  - Was passiert, wenn nach Aktion (S(50) ≥ T2(50) : Undo) und Einbringen der Seite S (flush (S)) ein Crash passiert?
  - Zustands-Logging und LIFO-Reihenfolge gewährleisten Idempotenz!

    ➥ Aber bei Übergangslogging? Manchmal allgemeinere Lösung erforderlich: Kompensation von Undo wird später eingeführt

7. In der graphischen Darstellung wird immer der persistente Zustand der DB (Seiten) und des Log (LSNs) gezeigt,

Crash-Recovery

- **Ziel:** Herstellung des jüngsten transaktionskonsistenten DB-Zustandes aus permanenter DB und temporärer Log-Datei

- **Bei Update-in-Place (Non-Atomic):**
  - Zustand der permanenten DB nach Crash unvorhersehbar („chaotisch“)

    ➥ nur physische und physiologische Logging-Verfahren anwendbar
  - Ein Block der permanenten DB ist entweder
    - aktuell oder
    - veraltet (NoForce) ➥ Redo oder
    - ‚schmutzig‘ (Steal) ➥ Undo

- **Bei Atomic:**
  - Permanente DB entspricht Zustand des letzten erfolgreichen Einbringens (Sicherungspunkt)
  - zumindest operationskonsistent (z. B. aktionskonsistent)

    ➥ **interne Operationen (Aktionen) oder gar DML-Befehle ausführbar (logisches Logging)**
  - **Force:** kein Redo
  - **NoForce:**

    a) transaktionskonsistentes Einbringen

    ➥ **Redo, jedoch kein Undo**

    b) aktionskonsistentes oder API-konsistentes Einbringen

    ➥ **Undo + Redo**

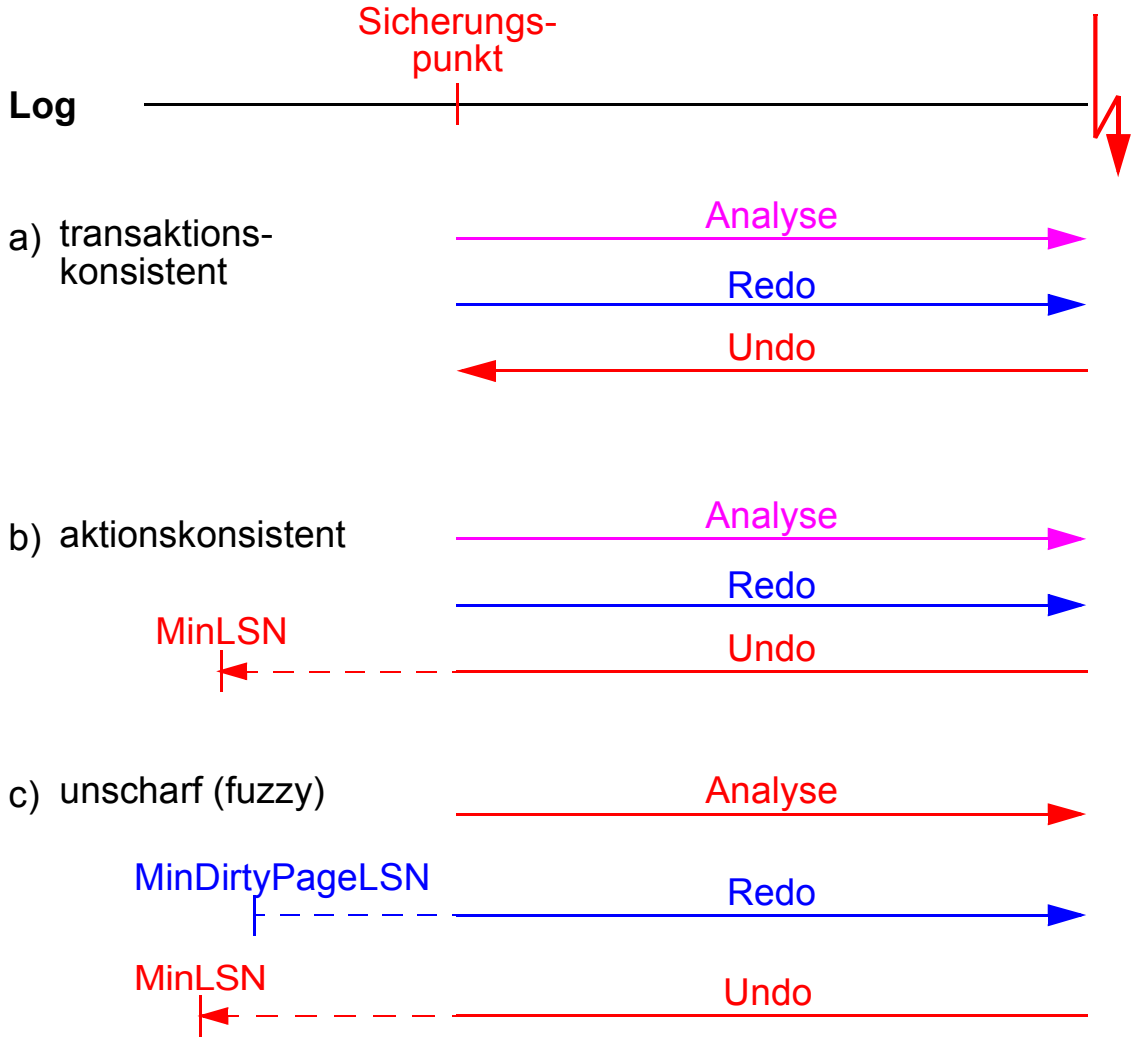
Allgemeine Restart-Prozedur

- **Temporäre Log-Datei wird 3-mal gelesen**

1. ***Analyse-Phase*** *(vom letzten Sicherungspunkt bis zum Log-Ende):*
   Bestimmung von ***Gewinner-*** und ***Verlierer-TA*** sowie der Seiten, die von ihnen geändert wurden

2. ***Redo-Phase:***
   Vorwärtslesen des Log: Startpunkt abhängig vom Sicherungspunkttyp:
   Selektives Redo bei Seitensperren (redo winners) oder
   Repeating History (vollständiges Redo) möglich

3. ***Undo-Phase:***
   Rücksetzen der Verlierer-TA durch Rückwärtslesen des Logs bis zum BOT-Satz der ältesten Verlierer-TA



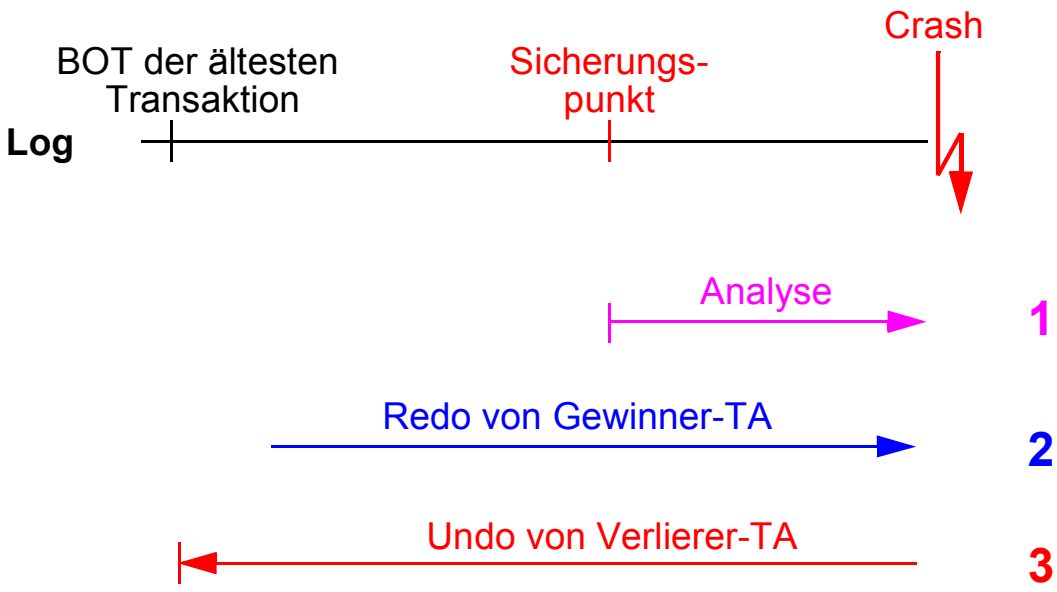
Restart-Prozedur bei Update-in-Place

- **Attribute: Non-Atomic, Steal, NoForce, Fuzzy Checkpoint**

1. ***Analyse-Phase*** *(vom letzten Sicherungspunkt bis zum Log-Ende):*

2. ***Redo-Phase:***
   Startpunkt abhängig vom Sicherungspunkttyp: hier MinDirtyPageLSN
   Selektives Redo: nur Wiederholung der Änderungen der Gewinner-TA

3. ***Undo-Phase:***
   Rücksetzen der Verlierer-TA bis MinLSN



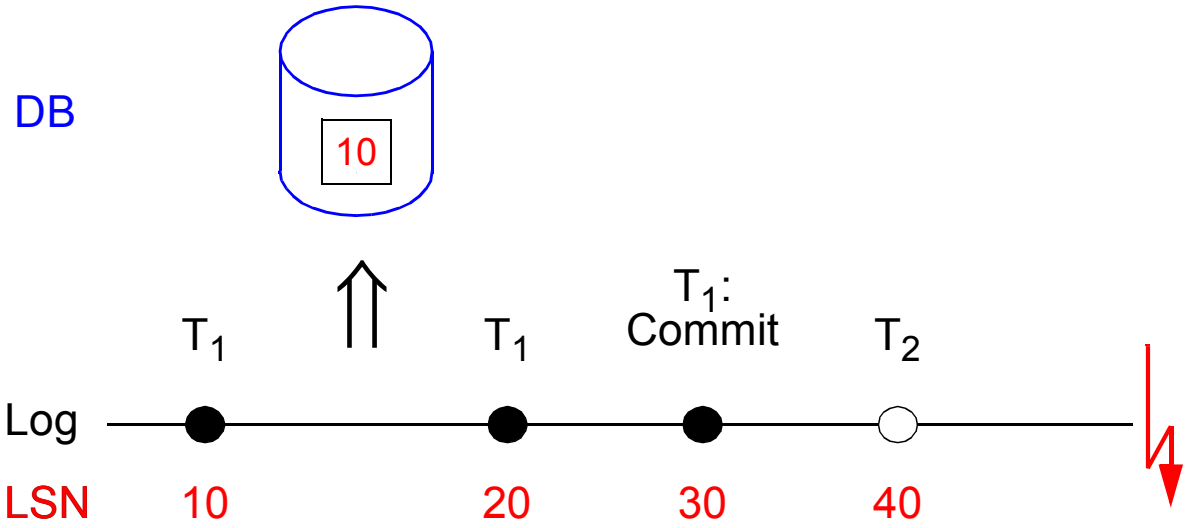
- **Aufwandsaspekte**
  - Für Schritt 2 und 3 sind betroffene DB-Seiten einzulesen
  - LSN der Seiten zeigen, ob Log-Informationen anzuwenden sind
  - Am Ende sind alle geänderten Seiten wieder auszuschreiben, bzw. es wird ein Sicherungspunkt erzeugt

Redo-Recovery

- **Physiologisches und physisches Logging:**
  Notwendigkeit einer Redo-Aktion für Log-Satz L wird über PageLSN der betroffenen Seite B angezeigt

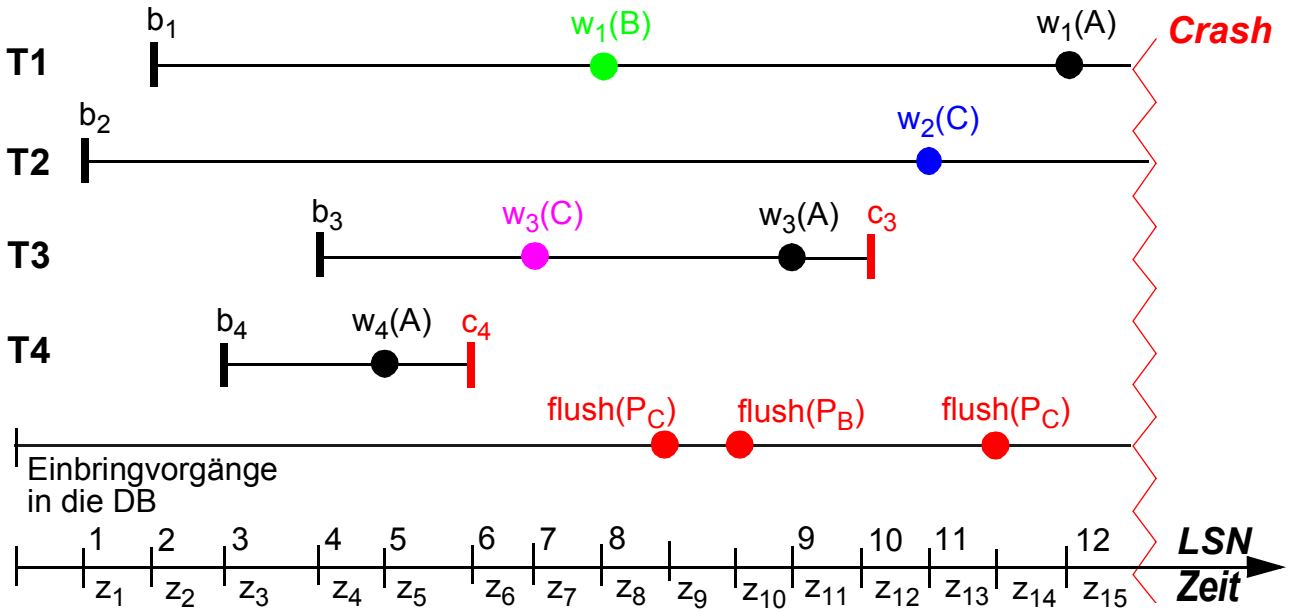
```
if (B nicht gepuffert) then (lies B in den Hauptspeicher ein);
if     LSN (L) > PageLSN (B) then do;
            Redo (Änderung aus L);
            PageLSN (B) := LSN (L);
end;
```

- **Wiederholte Anwendung des Log-Satzes**
  - (z. B. nach mehrfachen Fehlern) erhält Korrektheit (Redo-Idempotenz)
  - Wie verläuft die Recovery bei Crashs während des Restart?



  - Was passiert bei Crash im Restart nach T1(20) : Redo, S(20), wenn
    - Seite S eingebracht war (flush(S))
    - Seite S noch nicht eingebracht war?

Restart – Beispiel



Zeit	Aktion	Änderung im DB-Puffer (Seite, LSN)	Änderung in der DB (Seite, LSN)	Log-Puffer: (LSN, TAID, Log-Info, PrevLSN)	Log-Datei: zugefügte Einträge (LSNs)
z_1	b_2			1, T_2, BOT, 0	
z_2	b_1			2, T_1, BOT, 0	
z_3	b_4			3, T_4, BOT, 0	
z_4	b_3			4, T_3, BOT, 0	
z_5	$w_4(A)$	P_A, 5		5, T_4, U/R(A), 3	
z_6	c_4			6, T_4, EOT, 5	1, 2, 3, 4, 5, 6
z_7	$w_3(C)$	P_C, 7		7, T_3, U/R(C), 4	
z_8	$w_1(B)$	P_B, 8		8, T_1, U/R(B), 2	
z_9	flush(P_C)		P_C, 7		7, 8
z_{10}	flush(P_B)		P_B, 8		
z_{11}	$w_3(A)$	P_A, 9		9, T_3, U/R(A), 7	
z_{12}	c_3			10, T_3, EOT, 9	9, 10
z_{13}	$w_2(C)$	P_C, 11		11, T_2, U/R(C), 1	
z_{14}	flush(P_C)		P_C, 11		11
z_{15}	$w_1(A)$	P_A, 12		12, T_1, U/R(A), 8	

„We will meet again if your memory serves you well.” (Bob Dylan)

Restart – Beispiel (2)

- **Annahme:** Zu Beginn seien alle Seiten-LSNs auf Null gesetzt[8]
- **Analyse-Phase:** Gewinner-TA: T_3, T_4
  Verlierer-TA: T_1, T_2
  Relevante Seiten: P_A, P_B, P_C

  Im Restart-Beispiel ändert nie mehr als eine TA gleichzeitig in einer Seite, was einem Einsatz von Seitensperren entspricht. Deshalb ist **Selektives Redo**, also nur das Redo der Gewinner-TA, ausreichend.
- **Redo-Phase: Log-Sätze für T_3 und T_4 vorwärts prüfen**

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_4	P_A			
T_3	P_C			
T_3	P_A			

(Redo nur, wenn Seiten-LSN < Log-Satz-LSN)

➥ Seiten-LSNs wachsen monoton

- **Undo-Phase: Log-Sätze für T_1 und T_2 rückwärts prüfen**

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_1	P_A	9	12	Kein Undo, ohnehin nicht in Log-Datei
T_2	P_C			
T_1	P_B			

(Undo nur, wenn Seiten-LSN ≥ Log-Satz-LSN)

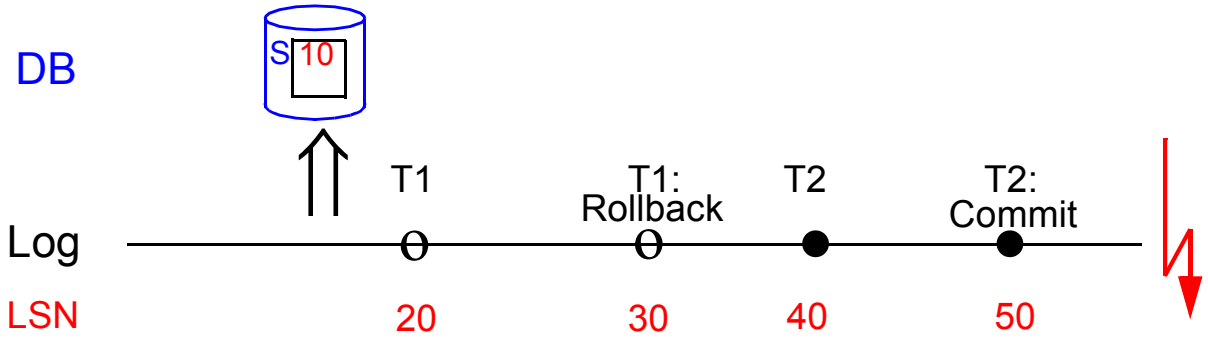
➥ Wegen der Seitensperren gibt es auf einer Seite keine Interferenz zwischen Redo- und Undo-Aktionen. Zustands-Logging sichert Undo-Idempotenz!

8. „This we know. All things are connected." (Chief Seattle)

Probleme bei LSN-Verwendung für Undo

- **Wiederholung der bisherigen Annahmen**
  - Selektives Redo (ist kosteneffektiv!)
  - Einsatz von Seitensperren
    (wenn erforderlich, Redo immer vor Undo in einer Seite)
  - Zustands-Logging

- **Problem 1: Rücksetzungen von TA**
  Bisherige LSN-Verwendung führt zu Problemen in der Undo-Phase
  bei vorherigem Rollback



Redo von T2:

S(10) < T2(40) : Redo, S(40)

Undo von T1:

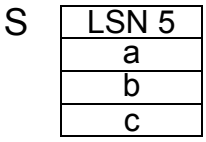
S(40) > T1(20) : Undo, **Fehler!**

- **Bemerkung:**
  - Es wird Änderung 20 zurückgesetzt, obwohl sie gar nicht in der Seite S vorliegt
  - Zuweisung von LSN = 20 zu S verletzt Monotonieforderung für Seiten-LSNs
    (Was passiert bei Crash nach Zuweisung?)

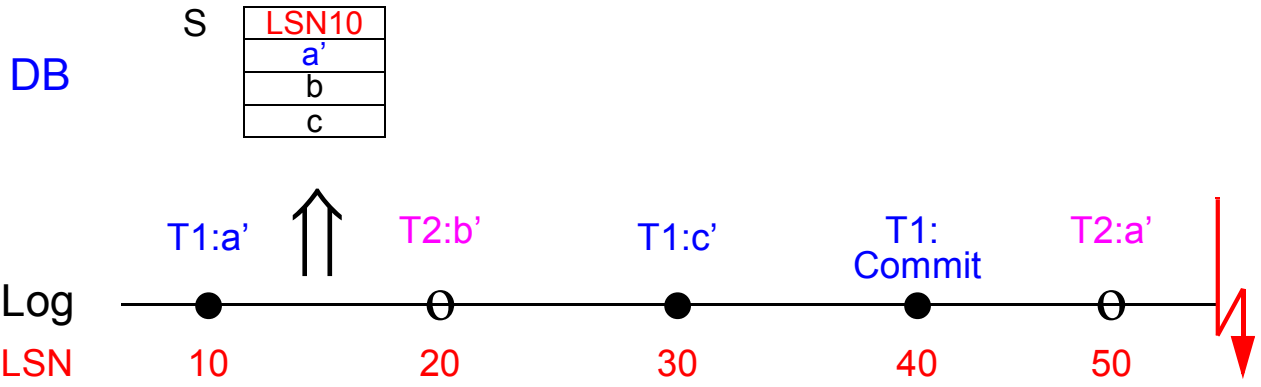
Probleme bei LSN-Verwendung für Undo (2)

- **Problem 2: Satzsperren**

  - Ausgangszustand der Seite S



  - T1 und T2 ändern gleichzeitig in Seite S



Redo von T1:

S(10) ≥ T1(10) : kein Redo

S(10) < T1(30) : Redo, S(30)

Undo von T2 (LIFO):

S(30) < T2(50) : kein Undo

S(30) > T2(20) : Undo, **Fehler!**

➥ **Allgemeinere Behandlung des Undo erforderlich !**

Fehlertoleranz des Restart

- **Forderung: Idempotenz des Restart**

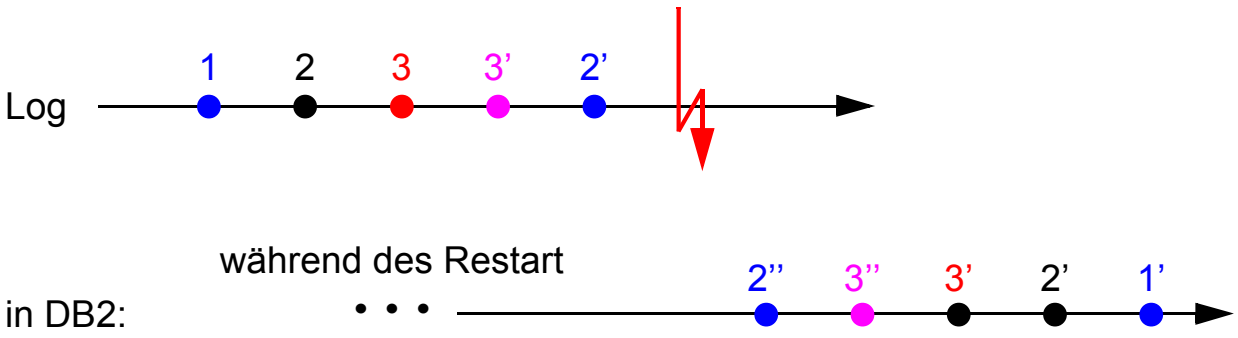
  Undo(Undo(. . . (Undo(A)) . . .)) = Undo(A)

  Redo(Redo(. . . (Redo(A)) . . .)) = Redo(A)

  - Idempotenz der Redo-Phase wird dadurch erreicht, dass LSN des Log-Satzes, für den ein Redo tatsächlich ausgeführt wird, in die Seite eingetragen wird.
    Redo-Operationen erfordern **keine zusätzliche** Protokollierung
  - Seiten-LSNs müssen monoton wachsen. Deshalb kann in der Undo-Phase nicht entsprechend verfahren werden.
  - Gewährleistung der Idempotenz der Undo-Phase erfordert ein neues Konzept: CLR = Compensation Log Record

- **Logging**

  - Änderungen der DB sind durch Log-Einträge abzusichern – und zwar im Normalbetrieb und beim Restart!
  - Was passiert im Fall eines Crash beim Undo?
    Aktionen 1-3 sollen zurückgesetzt werden: I' ist CLR für I und I'' ist CLR für I'



➥ Problem von kompensierenden Kompensationen!

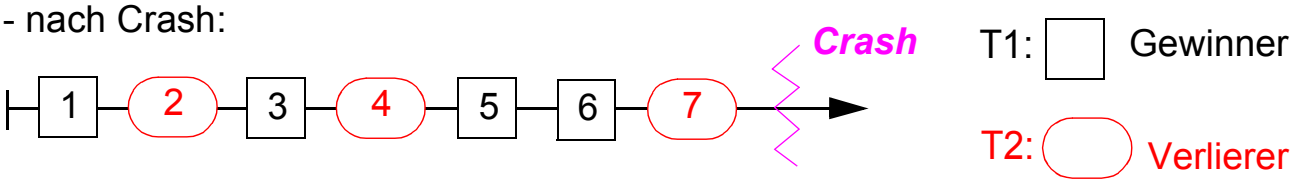
➥ Crash bei Restart!?

Compensation Log Records (CLR)

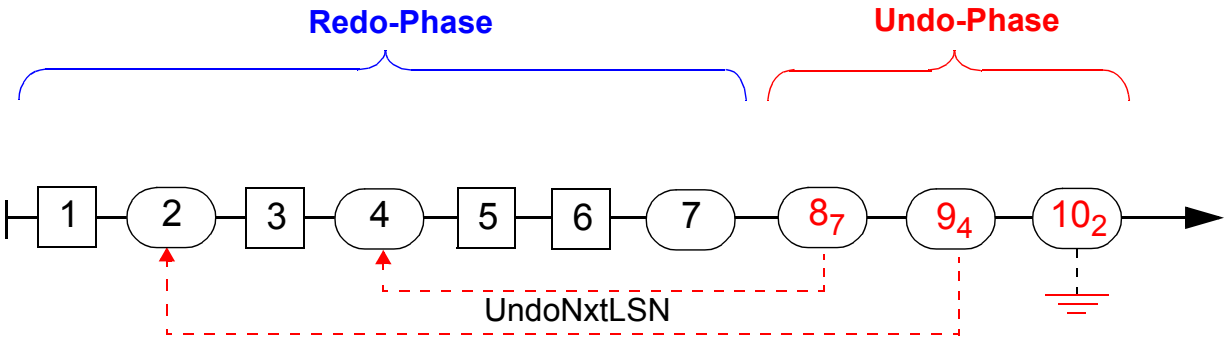
- **Optimierte Lösung**
  - Einsatz von CLRs bei allen Undo-Operationen: Rollback und Undo-Phase
  - in der Redo-Phase: **Repeating History** von Gewinnern und Verlierern (volständiges Redo)

- **Schematische Darstellung der Log-Datei**

- nach Crash:



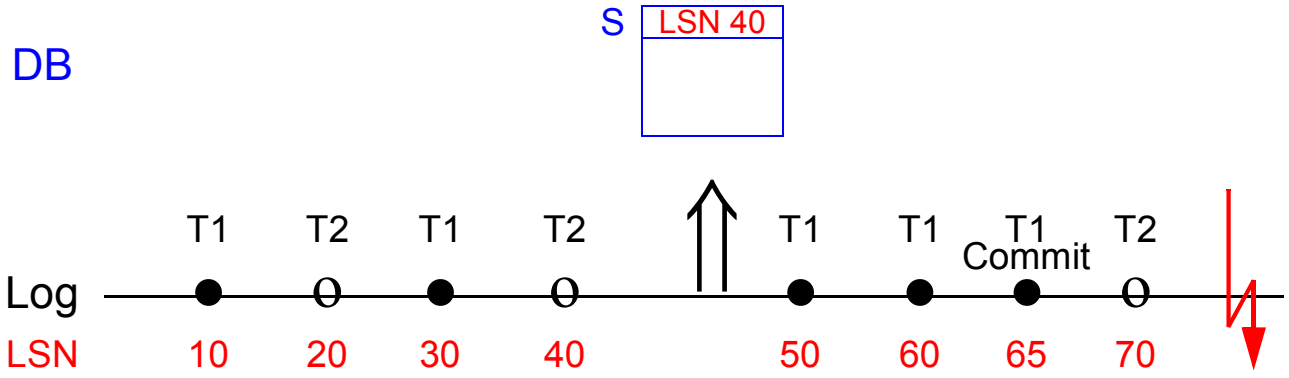
- nach vollständigem Restart:



  - **Die Redo-Information eines CLR** entspricht der während der **Undo-Phase ausgeführten Undo-Operation**
  - CLR-Sätze werden bei erneutem Restart benötigt (nach Crash beim Restart). Ihre Redo-Information wird während der **Redo-Phase** angewendet. Dabei werden Seiten-LSNs geschrieben.
    ➥ **Die Redo-Phase ist idempotent!**
  - CLRs benötigen keine Undo-Information, da sie während nachfolgender Undo-Phasen übersprungen werden (UndoNxtLSN)

CLR (2)

- **Detaillierung des Beispiels**
  - T1 Gewinner, T2 Verlierer
  - Alle Änderungen betreffen Seite S;
    es müssen also **Satzsperren** vorliegen
  - Zustand nach Crash 1:



Repeating History: S(40) >T1(10) : –

. . .

S(40) ≥T2(40) : –

S(40) < T1(50) : Redo, S(50)

S(50) <T1(60) : Redo, S(60)

S(60) < T2(70) : Redo, S(70)

Undo von T2:

CLR(80) : Kompensieren von T2(70), S(80)

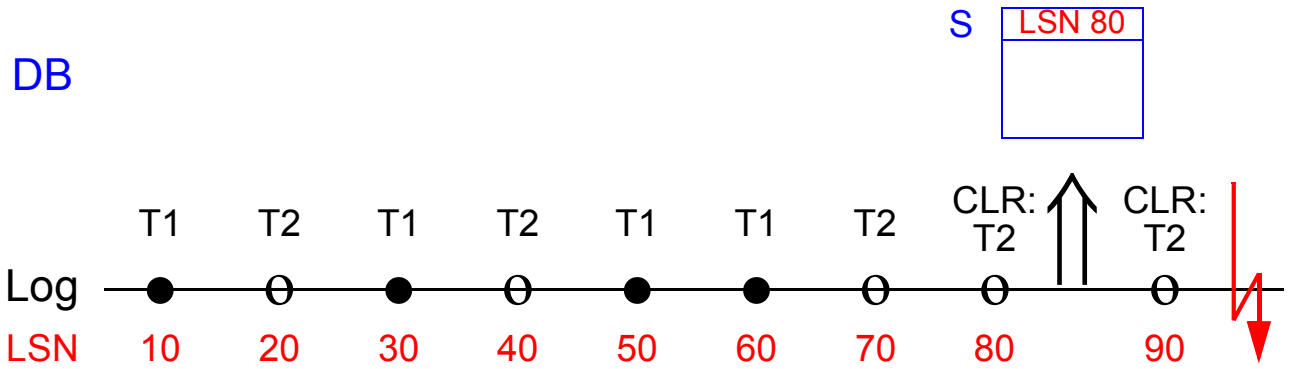
Schreiben von S in die DB (Flush S)

CLR(90) : Kompensieren von T2(40), S(90)

Crash

CLR (3)

- **Fortsetzung des Beispiels**
  - Zustand nach Crash 2:



Repeating History:

S(80) >T1(10)	:	–
. . .		
S(80) > T2(70)	:	–
CLR(80)	:	–
CRL(90)	:	Kompensieren von T2(40), S(90)

Undo von T2:

CLR(100)	:	Kompensieren von T2(20), S(100)

erfolgreiches Ende!

➥ Repeating History bei Einsatz von CLRs ist auch sicher bei Satzsperren!

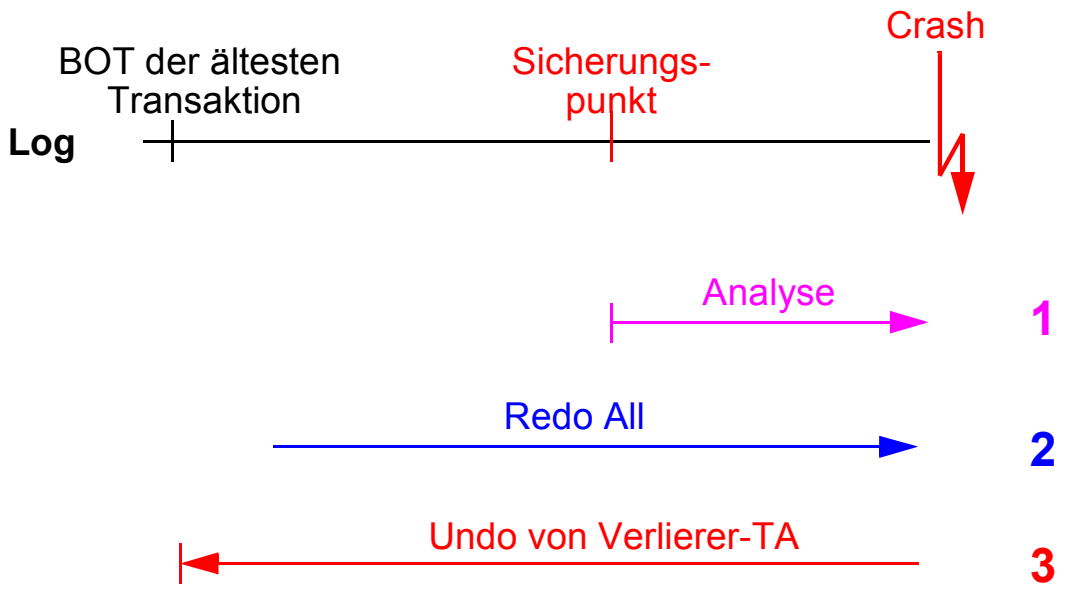
Restart-Prozedur bei Update-in-Place

- **Attribute: Non-Atomic, Steal, NoForce, Fuzzy Checkpoint**

1. ***Analyse-Phase*** *(vom letzten Sicherungspunkt bis zum Log-Ende):*

2. ***Redo-Phase:***
   Startpunkt abhängig vom Sicherungspunkttyp: hier MinDirtyPageLSN
   Repeating History: Wiederholung aller Änderungen in der DB
   (ggf. im DB-Puffer, auch die von Verlierer-TA), falls erforderlich

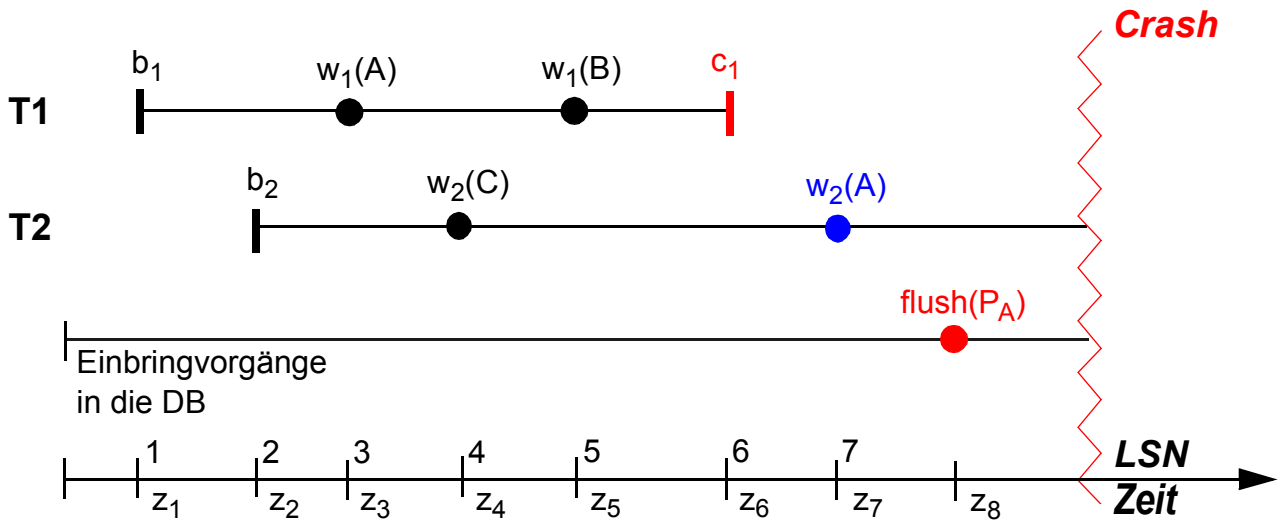
3. ***Undo-Phase:***
   Rücksetzen der Verlierer-TA bis MinLSN



- **Umsetzung durch ARIES[9]**
  (Algorithm for Recovery and Isolation Exploiting Semantics)
  - entwickelt von C. Mohan et al. (IBM Almaden Research)
  - realisiert in einer Reihe von kommerziellen DBS

9. C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging,* in: ACM Transactions on Database Systems 17:1, 1992, 94-162

Restart – Beispiel 2



Zeit	Aktion	Änderung im DB-Puffer (Seite, LSN)	Änderung in der DB (Seite, LSN)	Log-Puffer: (LSN, TAID, Log-Info, PrevLSN)	Log-Datei: zugefügte Einträge (LSNs)
z_1	b_1			1, T_1, BOT, 0	
z_2	b_2			2, T_2, BOT, 0	
z_3	$w_1(A)$	P_A, 3		3, T_1, U/R(A), 1	
z_4	$w_2(C)$	P_C, 4		4, T_2, U/R(C), 2	
z_5	$w_1(B)$	P_B, 5		5, T_1, U/R(B), 3	
z_6	c_1			6, T_1, EOT, 5	1, 2, 3, 4, 5, 6
z_7	$w_2(A)$	P_A, 7		7, T_2, U/R(A), 4	
z_8	flush(P_A)		P_A, 7		7

Restart – Beispiel 2 (2)

- **Annahme:** Zu Beginn seien alle Seiten-LSNs auf Null gesetzt
- **Analyse-Phase:**

  Gewinner-TA: T_1

  Verlierer-TA: T_2

  Relevante Seiten: P_A, P_B, P_C

  Im Restart-Beispiel 2 wird **Repeating History** durchgeführt.
  Zur Gewährleistung der Idempotenz der Undo-Operationen wird für jede ausgeführte Undo-Operation ein CLR mit folgender Struktur angelegt:
  [LSN, TAID, PageID, Redo, PrevLSN, UndoNextLSN]

- **Redo-Phase: Log-Sätze aller TA (T_1, T_2) vorwärts prüfen**

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_1	P_A	7	3	Kein Redo
T_2	P_C	0 --> 4	4	Redo
T_1	P_B	0 --> 5	5	Redo
T_2	P_A	7	7	Kein Redo

(Redo nur, wenn Seiten-LSN < Log-Satz-LSN)

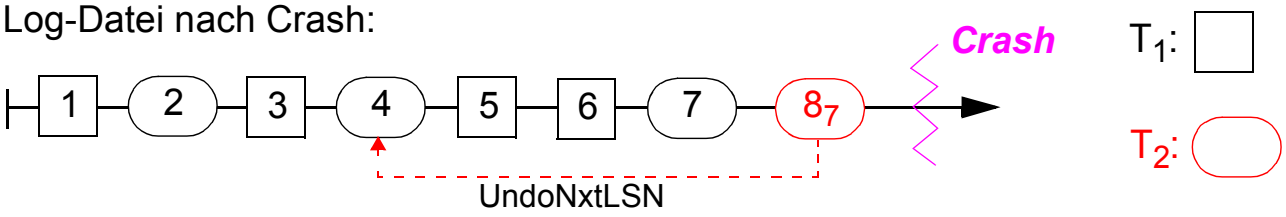
- **Undo-Phase:**
  Log-Sätze der Verlierer-TA T_2 sind rückwärts unabhängig von Seiten-LSN zu prüfen. Für jeden Log-Satz wird die zugehörige Undo-Operation durchgeführt und mit einem CLR in der Log-Datei vermerkt

TA	Log-Satz-LSN	Aktion
T_2	7	Undo und lege CLR [8, T_2, P_A, U(A), 7, 4] an
T_2	4	Undo und lege CLR [9, T_2, P_C, U(C), 8, 2] an
T_2	2	Undo und lege CLR [10, T_2, _ , _ , 9, 0] an

Restart – Beispiel 2 (3)

- **Annahme:** Crash während des Restart

  Log-Datei nach Crash:



- **Analyse-Phase:** dito

- **Redo-Phase:**

  Log-Sätze aller TA (T_1, T_2) inkl. CLRs vorwärts prüfen.

  Für jedes CLR wird jeweils Redo ausgeführt

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_1	P_A	7	3	Kein Redo
T_2	P_C	4	4	Kein Redo
T_1	P_B	5	5	Kein Redo
T_2	P_A	7	7	Kein Redo
T_2	P_A			Redo: mit U(A) kompensiert

- **Undo-Phase:**

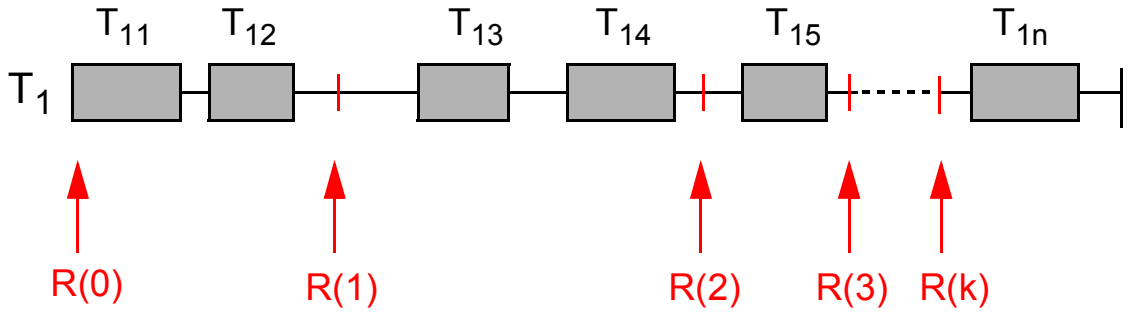
  Log-Sätze der Verlierer-TA T_2 (inkl. CLRs) sind rückwärts unabhängig von Seiten-LSN prüfen. Für jeden Log-Satz wird die zugehörige Undo-Operation durchgeführt und mit einem CLR in der Log-Datei vermerkt

TA	Log-Satz-LSN	Aktion
T_2	8	UndoNxtLSN = 4, dann weiter mit 4. Log-Satz (7. Log-Satz wird übersprungen, da er bereits mit dem 8. kompensiert wurde)
T_2	4	Undo und lege CLR [9, T_2, P_C, U(C), 8, 2] an
T_2	2	Undo und lege CLR [10, T_2, _ , _ , 9, 0] an

Zurücksetzen von Transaktionen

- **Transaktions-Recovery**
  - Zurücksetzen einer TA im laufenden DB-Betrieb
  - Nutzung der PrevLSN-Kette im temporären Log
  - Schreiben von optimierten CLRs, um mehrfaches Rücksetzen bei Restart zu vermeiden

- **Erweiterung zum partiellen Zurücksetzen innerhalb einer TA (R0)**
  - Voraussetzung: **transaktionsinterne Rücksetzpunkte** (*Savepoints*)



= atomarer Transaktionsschritt T_{1i}

R(i) = i-ter Rücksetzpunkt

$T_1 = (T_{11}, T_{12}, ..., T_{1n})$

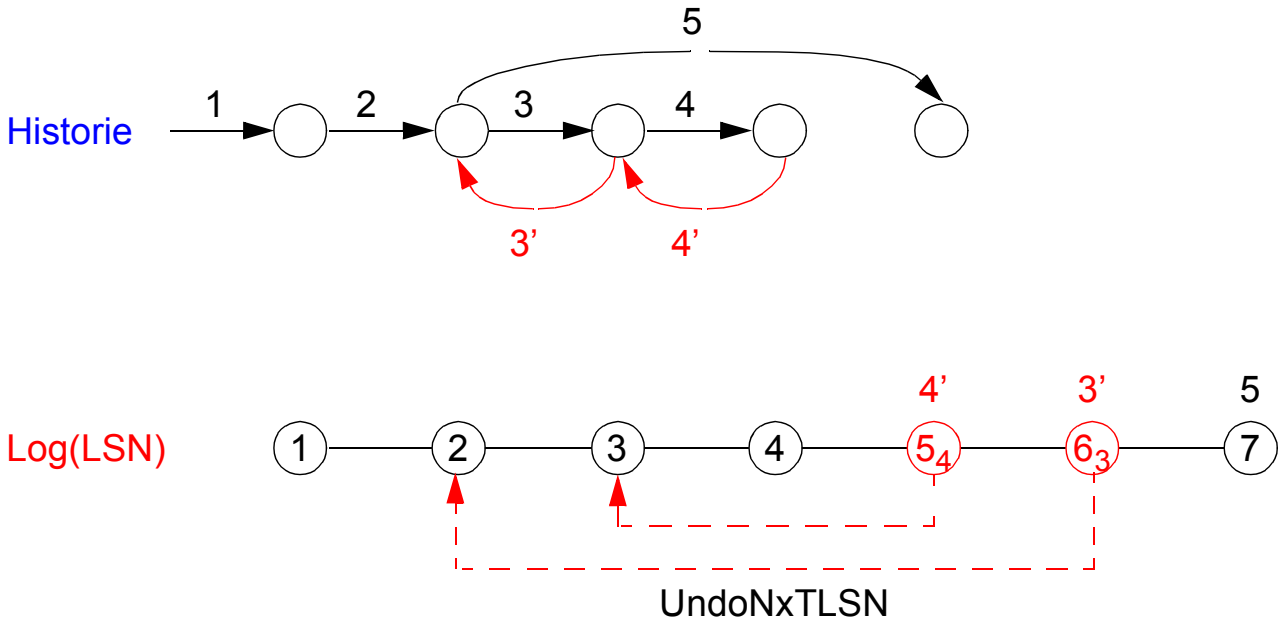
  - Zusätzliche Operationen: Save R(i)

    Restore R(j)

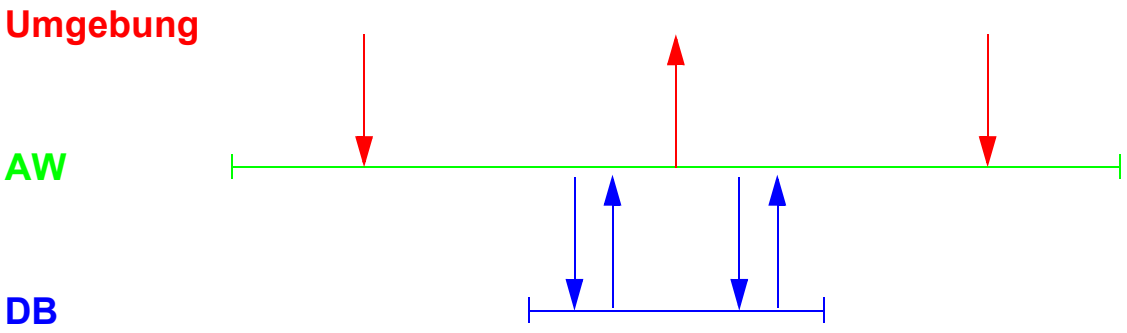
  - Protokollierung aller Änderungen, Sperren, Cursor-Positionen usw.
  - **Undo-Operation** bis R(j) in LIFO-Reihenfolge

Zurücksetzen von Transaktionen (2)

- **Partielles Rücksetzen einer TA**



- **Rücksetzpunkte** müssen vom DBS sowie vom Laufzeitsystem der Programmiersprache unterstützt werden
  - Derzeitige Implementierungen bieten keine Unterstützung von persistenten *Savepoints*!
  - Nach Systemfehler wird TA vollständig zurückgesetzt



Die Zehn Gebote[10]

Allgemeine Regeln

I. Recovery mit Übergangs-Logging setzt einen konsistenten DB-Zustand (bezüglich der protokollierten Operationen) zum Zeitpunkt der Recovery-Aktionen voraus.

II. Das Sperrgranulat muss mindestens so groß wie das Log-Granulat sein.

III. Nicht-atomare Einbringstrategien erfordern beim Restart erst Redo- bzw. Redo-All- (R2) und dann Undo-Recovery (R3). Bei atomaren Einbringstrategien ist dagegen die Reihenfolge von Undo- und Redo-Recovery unerheblich.

Regeln für Undo-Recovery

IV. Zustands-Logging erfordert ein WAL-Verfahren (wenn Seiten vor Commit in die DB eingebracht werden).

V. Für nicht-atomare Einbringstrategien ist Übergangs-Logging bei Redo- und Undo-Recovery generell nicht anwendbar.

VI. Wenn das Log-Granulat kleiner als die Transporteinheit (Blockgröße) des Systems ist, kann ein Systemfehler Medien-Recovery (R4) erfordern.

VII. Partielles Zurücksetzen innerhalb einer TA (R0) verletzt potenziell das 2PL-Protokoll (Programmierdisziplin erforderlich).

Regeln für Redo-Recovery

VIII. Log-Information für Redo-Recovery muss unabhängig von den Maßnahmen für Undo-Recovery gesammelt werden.

IX. Log-Information für Redo-Zwecke muss spätestens in Phase 1 von Commit geschrieben werden.

X. Um die Wiederholbarkeit der Ergebnisse aller Transaktionen bei der Redo-Recovery mit logischem Übergangs-Logging zu garantieren, müssen ihre Änderungen transaktionsweise (im Einbenutzer-Modus) im der ursprünglichen Commit-Reihenfolge nachvollzogen werden.

---

10. Härder, T., Reuter, A.: A Systematic Framework for the Description of Transaction-Oriented Logging and Recovery Schemes, Interner Bericht DVI 79-4, FG Datenverwaltungssysteme I, TH Darmstadt, Dez. 1979

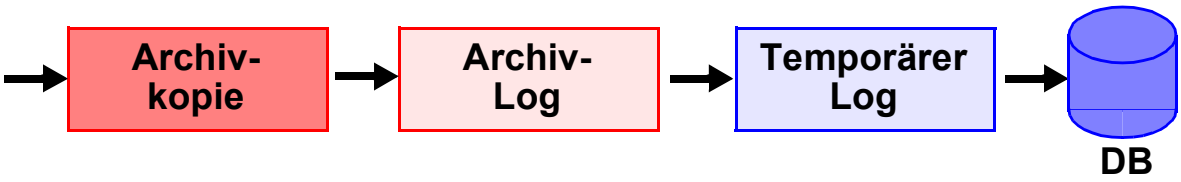
Medien-Recovery[11]

- **Spiegelplatten**
  - schnellste und einfachste Lösung
  - hohe Speicherkosten
  - Doppelfehler nicht auszuschließen

- **Alternative: Archivkopie + Archiv-Log**

- **Archivkopie + Archiv-Log sind längerfristig verfügbar zu halten**
  (auf Band)

  ➥ Problem von Alterungsfehlern
  - Führen von Generationen der Archivkopie
  - Duplex-Logging für Archiv-Log

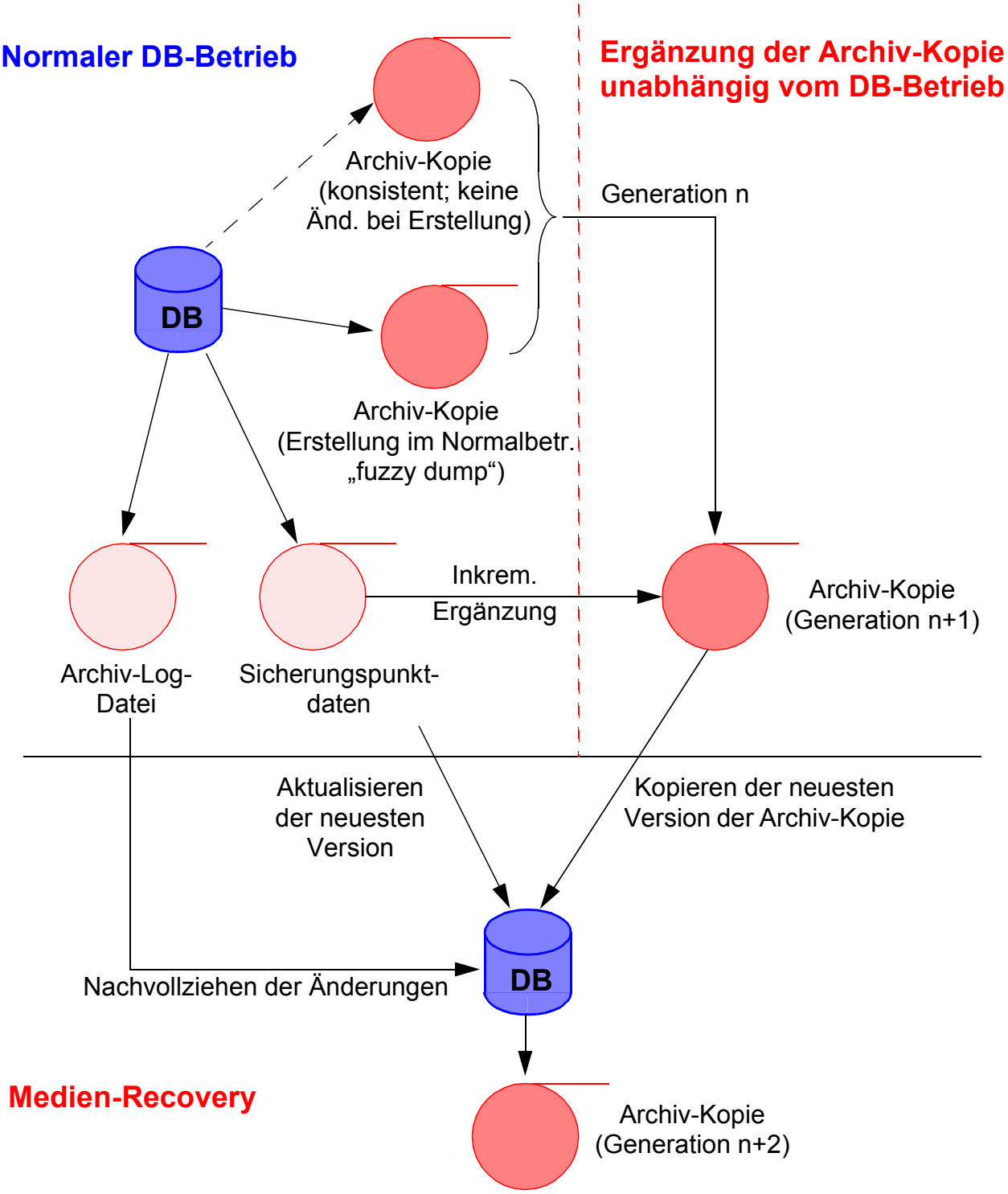


- **Ableitung von Archivdaten**
  - Sammlung sehr großer Datenvolumina als nachgelagerter Prozess
  - Archiv-Log kann offline aus temporärer Log-Datei abgeleitet werden
  - Erstellung von Archivkopien und Archiv-Log erfolgt segmentorientiert

11.„Don't worry, be happy." (Bobby McFerrin)

Medien-Recovery – Ein Szenarium

- **Komponenten der Medien-Recovery**



Optimierung der Erstellung der Archiv-Kopie durch inkrementelle Ergänzung mit Daten von Sicherungspunkten und ggf. Archiv-Log

Erstellung der Archivkopie

- **Anhalten des Änderungsbetriebs zur Erstellung einer DB-Kopie i. Allg. nicht tolerierbar**

- **Alternativen:**

  a) Incremental Dumping

  - Ableiten neuer Generationen aus „Urkopie“
  - nur Änderungen seit der letzten Archiv-Kopie protokollieren
  - Offline-Erstellung einer aktuelleren Kopie

  b) *Online-Erstellung einer Archivkopie*
  (parallel zum Änderungsbetrieb)

- **Unterschiedliche Konsistenzgrade:**

  b1) Fuzzy Dump

  - Kopieren der DB im laufenden Betrieb, kurze Lesesperren
  - bei Medienfehler Archiv-Log ab Beginn der Dump-Erstellung anzuwenden

  b2) Aktions- oder operationskonsistente Archivkopie
  (Voraussetzung bei Aktions- oder Operations-Logging (logisches Logging)

  b3) Transaktionskonsistente Archivkopie
  (Voraussetzung bei logischem Transaktions-Logging)

  - **Black-/White-Verfahren**
  - **Copy-on-Update-Verfahren**

Black-/White-Verfahren[12]

- **Spezieller Dump-Prozess zur Erstellung der Archiv-Kopie**

  ➥ **Erzeugung transaktionskonsistenter Archiv-Kopien**

- **Kennzeichnung der Seiten**
  - **Paint-Bit** pro Seite:
    - weiß: Seite wurde noch nicht überprüft
    - schwarz: Seite wurde bereits verarbeitet
  - **Modified-Bit** pro Seite zeigt an, ob eine Änderung seit Erstellung der letzten Archiv-Kopie erfolgte
  - Dump-Prozess färbt alle weißen Seiten schwarz und schreibt geänderte Seiten in Archiv-Kopie:

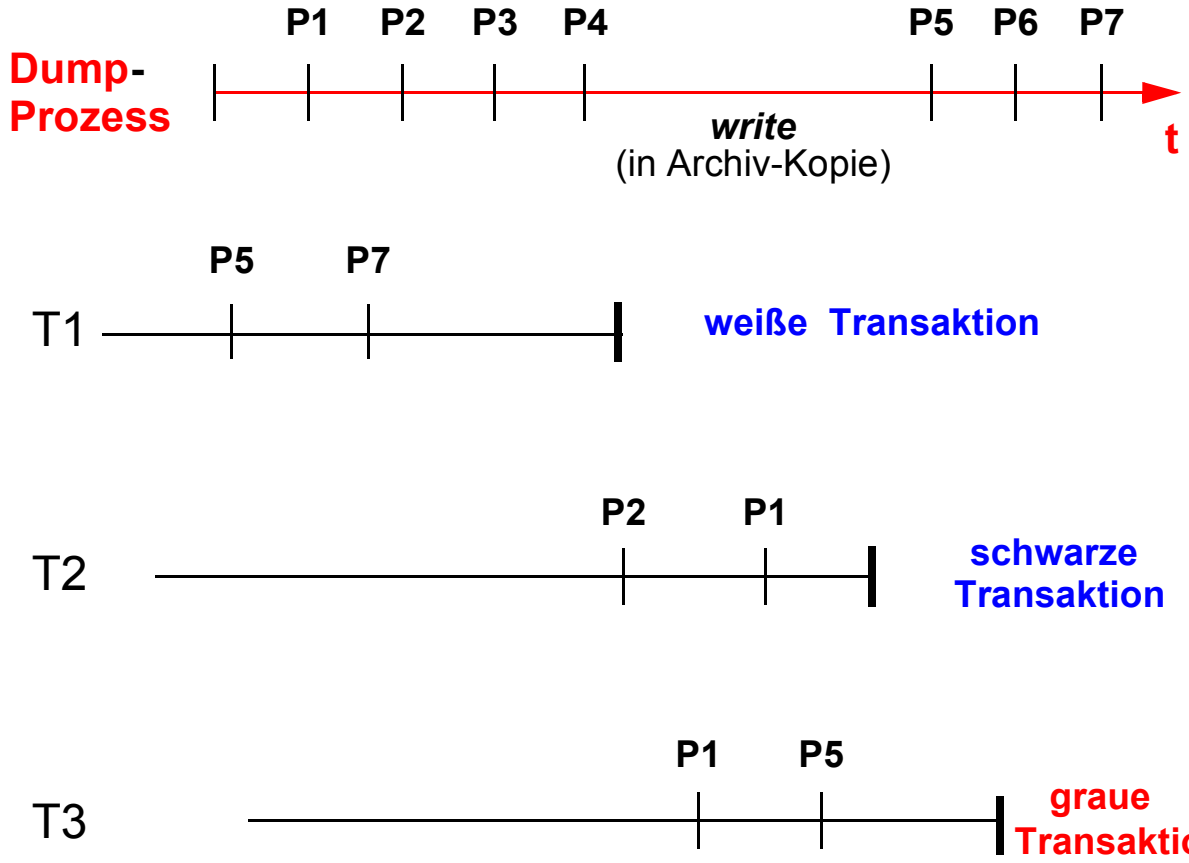
```
WHILE there are white pages  DO;
    lock any white page;
    IF page is modified THEN DO;
        write page to archive copy;
        clear modified bit;
    END;
    change page color;
    release page lock;
END;
```

- **Rücksetzregel**
  - Transaktionen, die sowohl weiße als auch schwarze Objekte geändert haben („graue Transaktionen“), werden zurückgesetzt
  - „Farbtest“ am Transaktionsende

12. C. Pu: On-the-Fly, Incremental, Consistent Reading of Entire Databases, in: Algorithmica, 1986, 271- 287

Black-/White-Verfahren (2)

- Beispiel



Black-/White-Verfahren – Erweiterungen zur Vermeidung von Rücksetzungen

- **Turn-White-Strategien** (Turn gray transactions white)
  - Für graue Transaktionen werden Änderungen „schwarzer“ Objekte nachträglich in Archiv-Kopie geschrieben
  - Problem: transitive Abhängigkeiten
  - **Alternative:** alle Änderungen schwarzer Objekte seit Dump-Beginn werden noch geschrieben (repaint all)
  - Problem: Archiv-Kopie-Erstellung kommt u. U. nie zu Ende

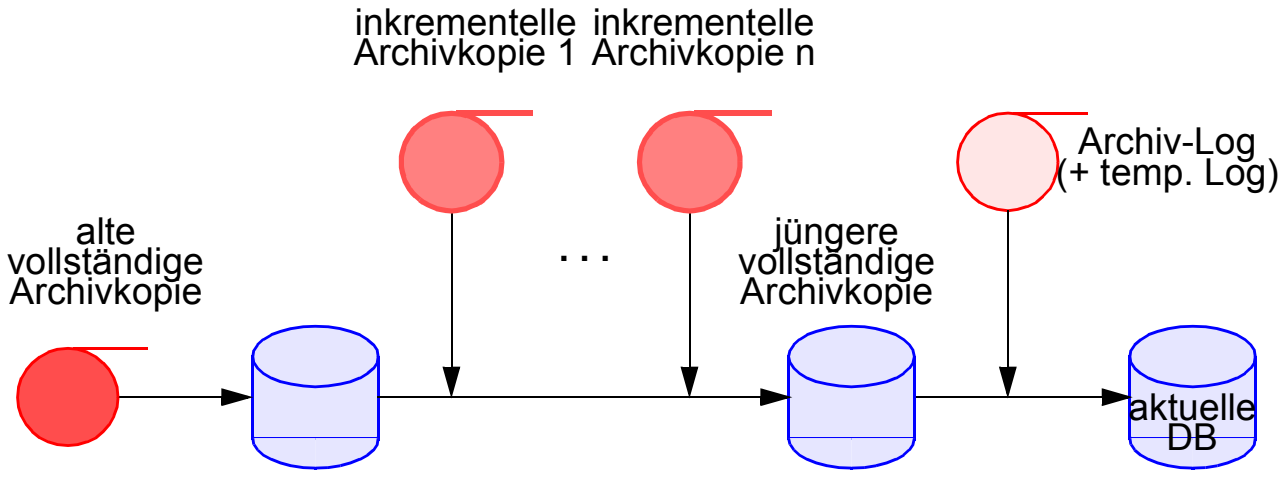
- **Turn-Black-Strategien**
  - Während der Erstellung einer Archiv-Kopie werden keine Zugriffe auf weiße Objekte vorgenommen
  - ggf. zu warten, bis Objekt gefärbt wird

- **Alternative: *Copy-on-Update* („save some")**
  - Während der Erstellung einer Archiv-Kopie wird bei Änderung eines weißen Objektes Kopie mit Before-Image der Seite angelegt
  - Dump-Prozess greift auf Before-Images zu
  - Archiv-Kopie entspricht DB-Schnappschuss bei Dump-Beginn

    ➥ wird in einigen DBS eingesetzt (DEC RDB)

Inkrementelles Dumping

- **Nur DB-Seiten, die seit der letzten Archivkopie-Erstellung geändert wurden, werden archiviert**



- **Erkennung geänderter Seiten**
  - Archivierungs-Bit pro Seite ➥ sehr hoher E/A-Aufwand
  - besser: Verwendung separater Datenstrukturen (Bitlisten)

- **Setzen eines Änderungsbits, falls**

  (PageLSN der ungeänderten Seite) < (LSN zu Beginn des letzten Dumps)

Zusammenfassung

- **Fehlerarten:**
  Transaktions-, System-, Gerätefehler und Katastrophen

- **Breites Spektrum von Logging- und Recovery-Verfahren**
  - Logging kann auf verschiedenen Systemebenen angesiedelt werden
  - erfordert ebenenspezifische Konsistenz im Fehlerfall
  - Physiologisches Logging/Aktions-Logging ist Seiten-Logging überlegen; in vielen DBS findet sich das **physiologische Logging** (flexiblere Recovery in einer DB-Seite, geringerer Platzbedarf, weniger E/As, Gruppen-Commit)

- **Synchronisationsgranulat muss größer oder gleich dem Log-Granulat sein**

- **Atomic-Verfahren**
  - erhalten den DB-Zustand des letzten Sicherungspunktes
  - gewährleisten demnach die gewählte Aktionskonsistenz auch bei der Recovery von einem Crash und
  - erlauben folglich logisches Logging

- **Update-in-Place-Verfahren**
  - sind i. Allg. Atomic-Strategien vorzuziehen, weil sie im Normalbetrieb wesentlich billiger sind und
  - nur eine geringe Crash-Wahrscheinlichkeit zu unterstellen ist
  - Sie erfordern jedoch physisches oder physiologisches Logging
  -
  -
  -

Zusammenfassung (2)

- **Grundprinzipien bei Update-in-Place**

1. WAL-Prinzip: Write Ahead Log für Undo-Info
2. Redo-Info ist spätestens bei Commit zu schreiben

- **Grundprinzipien bei Atomic**

1. WAL-Prinzip bei verzögertem Einbringen:

   TA-bezogene Undo-Info ist vor Sicherungspunkt zu schreiben
2. Redo-Info ist spätestens bei Commit auf die Log-Datei zu schreiben

- **NoForce-Strategien**
  - sind Force-Verfahren vorzuziehen
  - erfordern den Einsatz von Sicherungspunktmaßnahmen zur Begrenzung des Redo-Aufwandes:

    ➥ **„Fuzzy Checkpoints“** erzeugen den geringsten Overhead im Normalbetrieb

- **Steal-Methoden**
  - verlangen die Einhaltung des WAL-Prinzips
  - erfordern Undo-Aktionen nach einem Rechnerausfall

- **Idempotenz des Restart**
  - Operationen der Redo-Phase, falls erforderlich, erhöhen die Seiten-LSNs; Notwendigkeit der Wiederholung kann jederzeit erkannt werden
  - Idempotenz für Undo und Rollback durch Einführung von CLRs; nach Crash in der Undo-Phase werden Undo-Operationen beim nachfolgenden Restart in der Redo-Phase kompensiert (Erhöhung der Seiten-LSNs, beliebig oft unterbrechbar)

- **Erstellung von Archiv-Kopien:**
  „Fuzzy Dump” oder „Copy on Update” am besten geeignet