6. Logging und Recovery*

- **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**
- **Transaktions-Recovery**
- **Crash-Recovery**
  - Allgemeine Restart-Prozedur
  - Einsatz von Compensation Log Records
  - Restart-Bespiel
- **Medien-Recovery**

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

DB-Recovery

- **Aufgabe des DBVS:**
  Automatische Behandlung aller erwarteten Fehler
- **Voraussetzung:**
  Sammeln redundanter Informationen während des normalen Betriebes (*Logging*)
- **Fehlermodell von zentralisierten DBVS**
  - Transaktionsfehler
  - Systemfehler
  - Gerätefehler
- **Probleme**
  - Fehlererkennung
  - Fehlereingrenzung
  - Abschätzung des Schadens
  - Durchführung der Recovery
- **“A recoverable action is 30% harder and requires 20% more code than a non-recoverable action” (J. Gray)**

DB-Recovery (2)

- **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 Integritätsbedingungen 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)

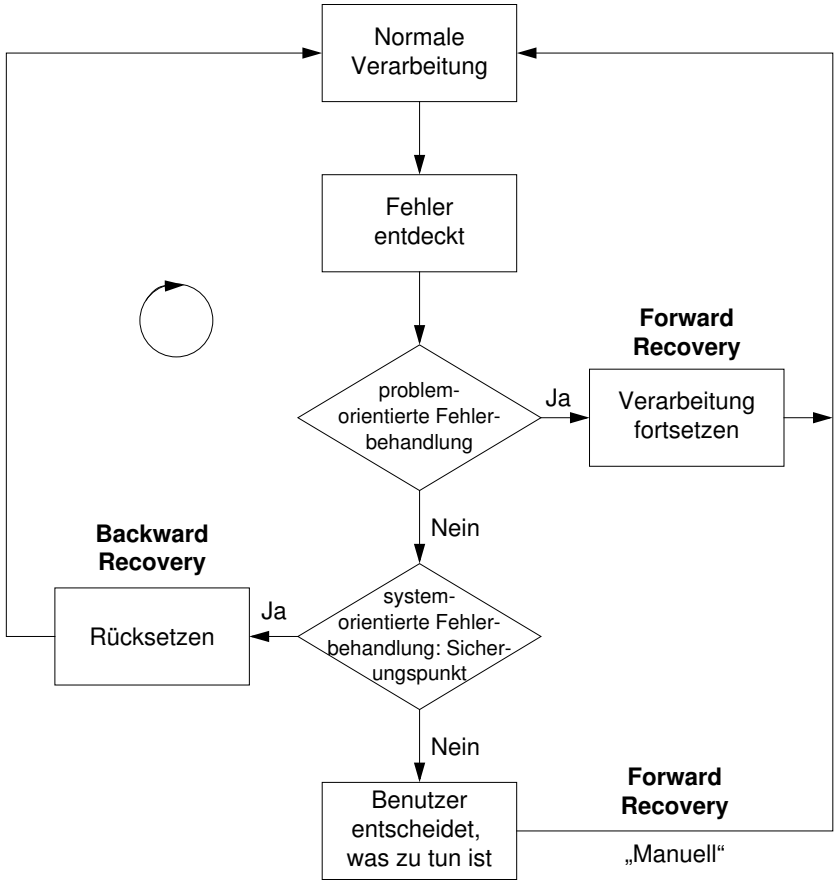
- **Forward-Recovery i. allg. nicht anwendbar**
  - Fehlerursache häufig falsche Programme, Eingabefehler u. ä.
  - durch Fehler unterbrochene Transaktionen sind zurückzusetzen (*Backward Recovery*)

- **Backward-Recovery**

  setzt voraus, daß auf allen Abstraktionsebenen genau definiert ist, auf welchen Zustand die DB im Fehlerfall zurückzusetzen ist.

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)	- Systemzusammenbruch mit Verlust der Hauptspeicherinhalte • Hardware-Fehler • falsche Werte in kritischen Tabellen	Systemfehler
	- Zerstörung von Sekundärspeichern	Gerätefehler
	- Zerstörung des Rechenzentrums	Katastrophen

Recovery-Arten

1. Transaktions-Recovery

Zurücksetzen einzelner (noch nicht abgeschlossener) Transaktionen im laufenden DB-Betrieb (Transaktionsfehler, Deadlock, etc.)

- vollständiges Zurücksetzen auf Transaktionsbeginn (TA-UNDO) bzw.
- partielles Zurücksetzen auf Rücksetzpunkt (*Savepoint*) innerhalb der Transaktion

2. Crash-Recovery nach Systemfehler

Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustandes:

- (partielles) REDO für erfolgreiche Transaktionen (Wiederholung verlorengegangener Änderungen)
- UNDO aller durch Ausfall unterbrochenen Transaktionen (Entfernen der Änderungen aus der permanenten DB)

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 der Basis gesicherter Archivkopien (Datenverlust!)

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 if 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!???

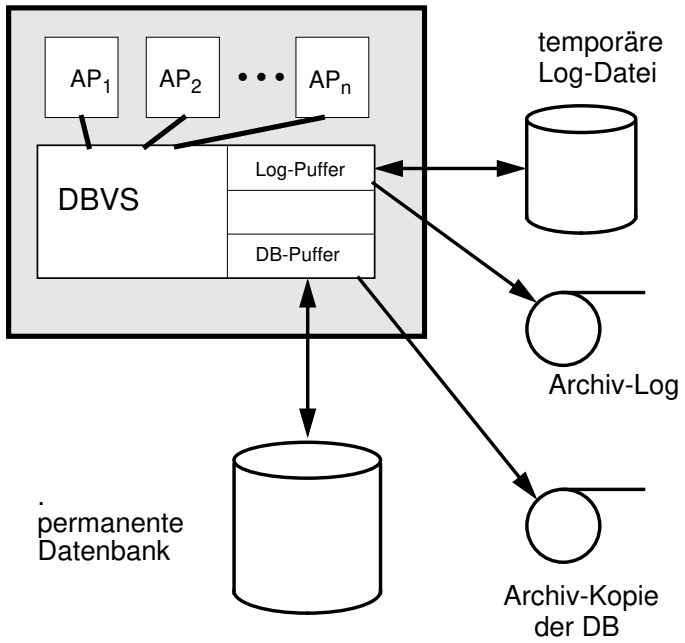
- **Annahmen**
  (Unter welchen Voraussetzungen funktioniert die Wiederherstellung der Daten?)
  - quasi-stabiler Speicher
  - fehlerfreier DBVS-Code
  - fehlerfreie Log-Daten
  - Durchführung der Recovery

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

- **Was ist zu tun, wenn . . .?**

* J. Gray (1998 Turing Lecture):
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!
Today's uptime: 99% for web sites, 99.99% for well managed sites (out 50 minutes/year)

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

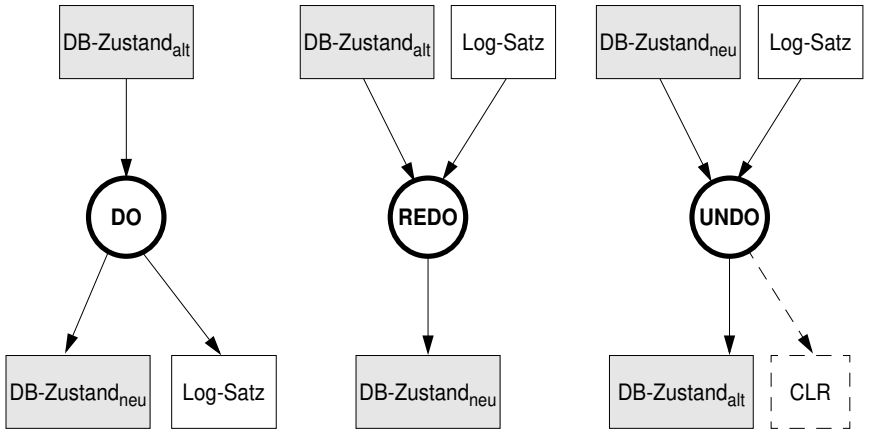
   DB + temp. Log ⇒ DB

2. **Behandlung von Gerätefehlern:**

   Archiv-Kopie + Archiv-Log ⇒ DB

Logging-Aufgaben

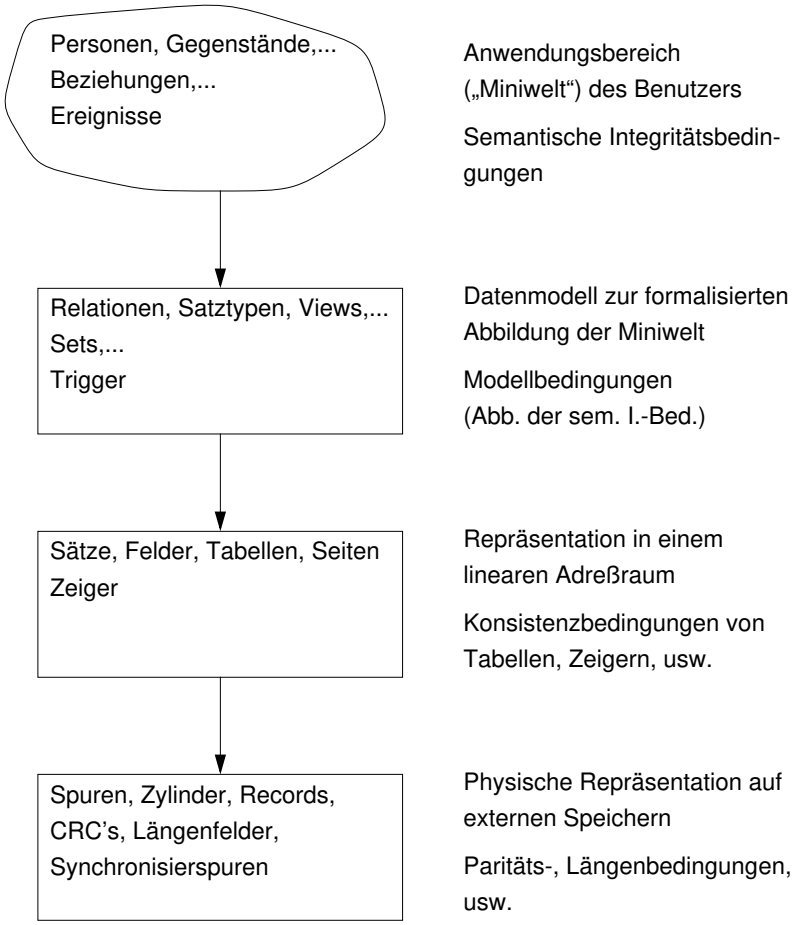
- **Logging**
  - Sammlung redundanter Daten bei Änderungen im Normalbetrieb (Do) als Voraussetzung für Recovery
  - Einsatz im Fehlerfall (Undo-, Redo-Recovery)
- **Do-Redo-Undo-Prinzip**



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

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

Abstraktionsebenen und Logging

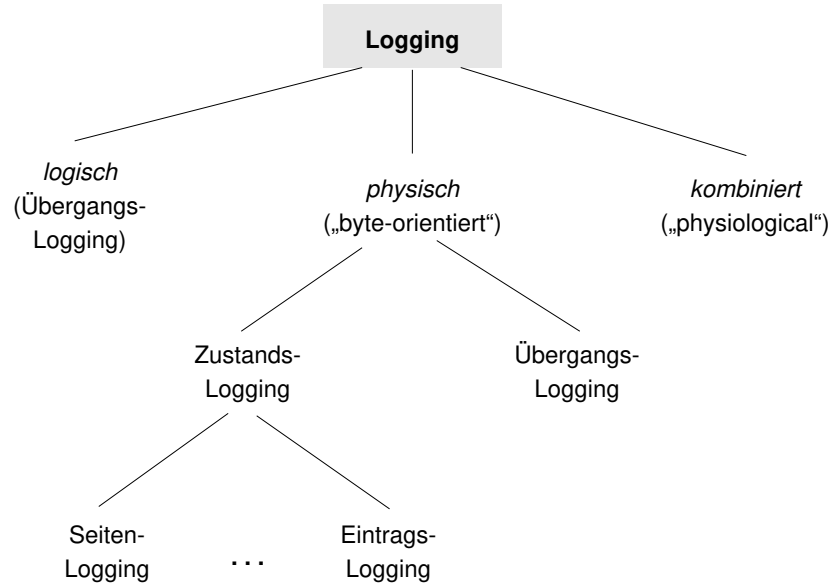


Logging kann auf jeder Ebene erfolgen:

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

➥ Wie kann ebenenspezifische Konsistenz (Aktions- oder Operationskonsistenz) im Fehlerfall garantiert werden ?

Klassifikation von Logging-Verfahren



- **Logisches Logging**
  - Protokollierung der ändernden DML-Befehle mit ihren Parametern
  - Generelles Problem:
    mengenorientierte Aktualisierungsoperation (z. B. DELETE <relation>)
  - UNDO-Probleme v.a. bei nicht-relationalen Systemen
    (z. B. Löschen einer Hierarchie von Set-Ausprägungen (ERASE ALL))
  - Voraussetzung:
    Nach einem Systemausfall müssen auf der permanenten Datenbank DML-Operationen ausführbar sein, d.h. sie muß wenigstens speicherkonsistent sein (Aktionskonsistenz)

    ➥ ***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

- **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: $A_1 \overset{1.}{\rightarrow} A_2 \overset{2.}{\rightarrow} A_3$

	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:

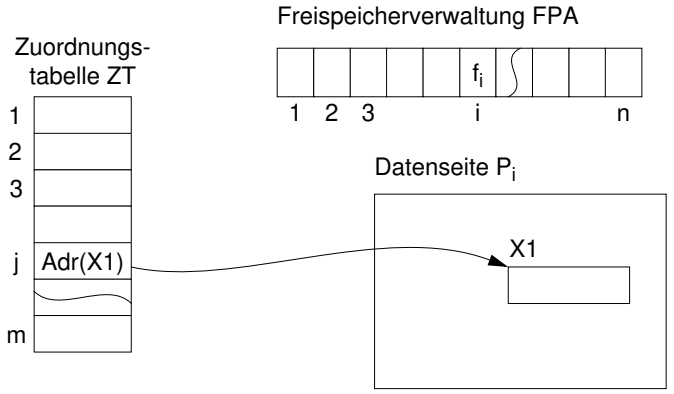
  A_1 als Anfangs- oder A_3 als Endzustand seien verfügbar

  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 Zustandslogging

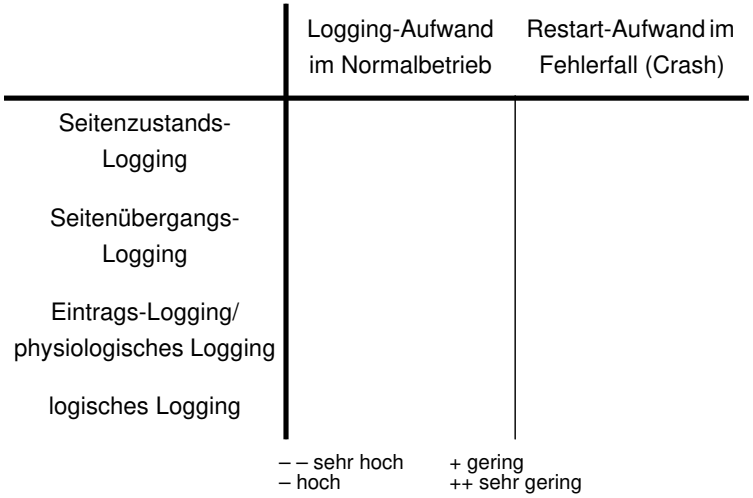
- **Einfachste Form der Implementierung: Page-Logging**



- **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



- **Vorteile von *Eintrags-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

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'))
  - Checkpoint-Sätze

- **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 Transaktion, 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 Transaktion. 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]

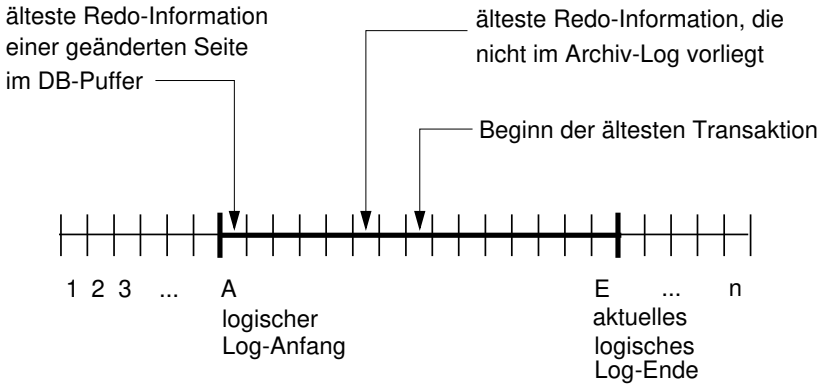
Aufbau der (temporären) Log-Datei (2)

- **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 Transaktionen 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



Nutzung von LSNs

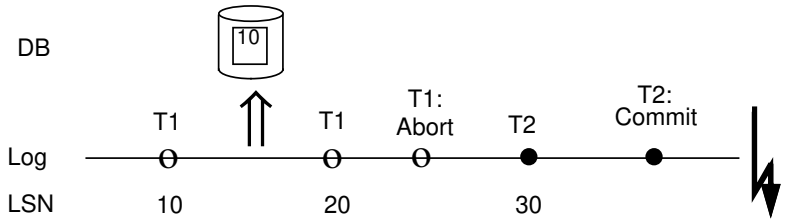
- **Seitenkopf von DB-Seiten enthält Seiten-LSN**
  - Die „Herausforderung“ besteht darin, beim Wiederanlauf zu entscheiden, ob für die Seite Recovery-Maßnahmen anzuwenden sind oder nicht (ob man den alten oder bereits den geänderten Zustand auf dem Externspeicher vorgefunden hat)
  - Dazu wird auf jeder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

- **Entscheidungsprozedur:**
  - **REDO *ist nur erforderlich, wenn***

    Seiten-LSN < LSN des Redo-Log-Satzes

  - **UNDO *ist nur erforderlich, wenn***

    Seiten-LSN ≥ LSN des Undo-Log-Satzes

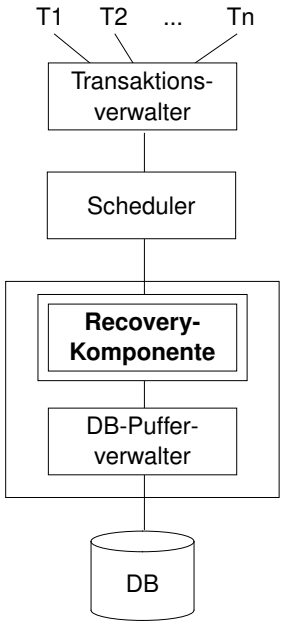


UNDO von Änderung 10

REDO von Änderung 30

Abhängigkeiten zu anderen Systemkomponenten

- **Stark vereinfachtes Modell**

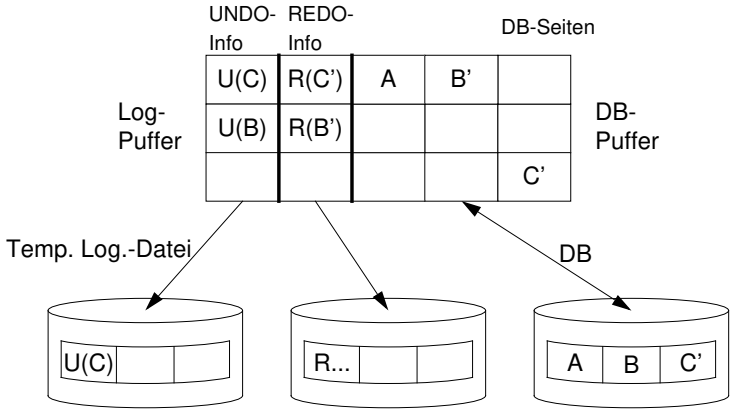


1. **Einbringstrategie für Änderungen**
   - direkt (NON-ATOMIC, *Update-in-Place*)
   - verzögert (ATOMIC, Bsp.: Schattenspeicherkonzept)
2. **DB-Pufferverwaltung**
   - Verdrängen ‚schmutziger‘ Seiten (STEAL vs. NOSTEAL)
   - Ausschreibstrategie für geänderte Seiten (FORCE vs. NOFORCE)
3. **Sperrverwaltung**

   (Wahl des Sperrgranulats)

Abhängigkeiten zur Einbringstrategie

- **Direkt (*Update-in-Place*)**
  - Geänderte Seite wird immer in denselben Block auf Platte zurückgeschrieben
  - ‚Atomares' Zurückschreiben mehrerer geänderter Seiten ist nicht möglich **(NON-ATOMIC**)



UNDO- und REDO-Info typischerweise in einer sequentiellen Datei

Es sind 2 Prinzipien einzuhalten (Minimalforderung):

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

   U(B) vor B'

2. **Ausschreiben der REDO-Info spätestens bei COMMIT**

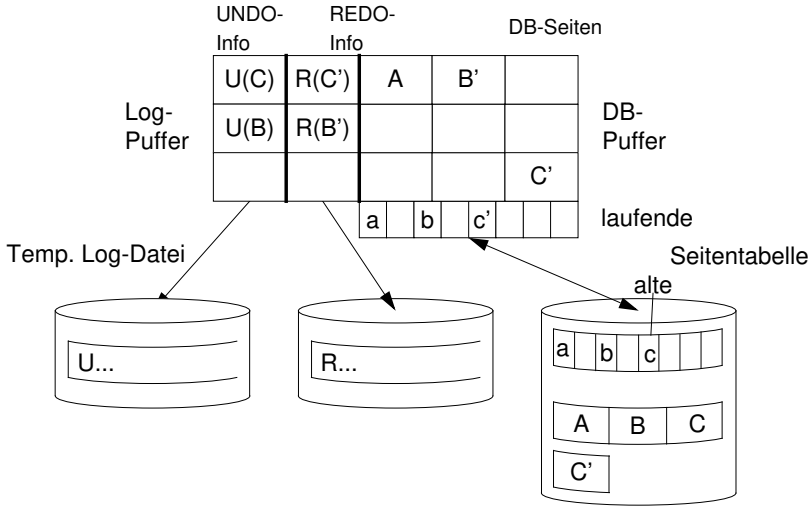
   R(C') + R(B') vor COMMIT

Abhängigkeiten zur Einbringstrategie (2)

- **Verzögert (ATOMIC)** (System R, SQL/DS)
  - Geänderte Seite wird in separaten Block auf Platte geschrieben
  - Seitentabelle gibt aktuelle Adresse einer Seite an
  - Verzögertes, atomares Einbringen mehrerer Änderungen ist durch Umschalten von Seitentabellen möglich

  ➥ *aktions- oder transaktionskonsistente DB auf Platte*

  (logisches Logging anwendbar)



1. **WAL-Prinzip bei verzögertem Einbringen**

   Transaktionsbezogene 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 Transaktion, ersetzt und in die permanente DB eingebracht werden

  \+ große Flexibilität zur Seitenersetzung

  – UNDO-Recovery vorzusehen
  (Transaktionsabbruch, Systemfehler)

  ➥ 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:**

  - Seiten mit schmutzigen Änderungen dürfen nicht ersetzt werden
  - Es ist keine UNDO-Recovery auf der permanenten DB vorzusehen
  - Probleme bei langen Änderungstransaktionen

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 Änderungstransaktionen

- **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 Transaktion ausgeführt werden kann, sind für ihre Änderungen ausreichende REDO-Informationen (z. B. *After-Images*) zu sichern

Weitere Abhängigkeiten

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

	Steal	**Nosteal**
Force		
Noforce		

- **Abhängigkeit zur Sperrverwaltung**

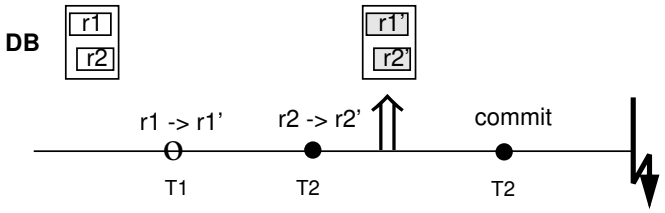
  Log-Granulat muß kleiner oder gleich dem Sperrgranulat sein!

  Beispiel:

  Sperren auf Satzebene,

  Before- bzw. *After-Images* auf Seitenebene

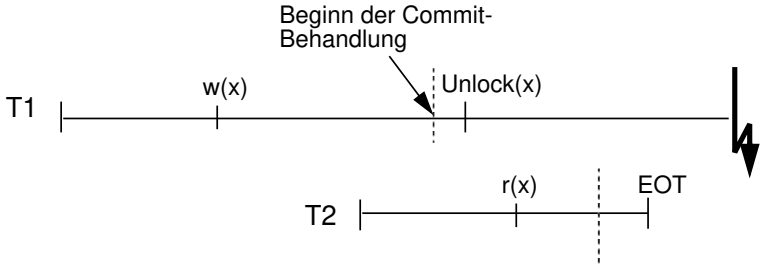
  ➥ UNDO (REDO) einer Änderung kann parallel durchgeführte Änderungen derselben Seite überschreiben *(lost update)*



Commit-Behandlung

- **Änderungen einer Transaktion sind vor Commit zu sichern**

- Andere Transaktionen dürfen Änderungen erst sehen, wenn Durchkommen der ändernden Transaktion gewährleistet ist (Problem der ‘Cascading Aborts‘)



- ***Zweiphasige Commit-Bearbeitung***

  Phase 1: Wiederholbarkeit der Transaktion 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 Transaktion 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. 60 sequentielle Schreibvorgänge pro Sekunde (1 Platte)

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

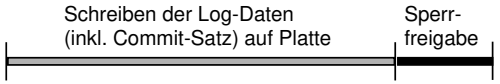
- **Gruppen-Commit**
  erlaubt Reduktion auf 0.1 - 0.2 Log-E/As pro Transaktion
  ➥ Durchsatzverbesserung v.a. bei Log-Engpaß oder hoher CPU-Auslastung

- Einsparung an CPU-Overhead für E/A reduziert CPU-Wartezeiten

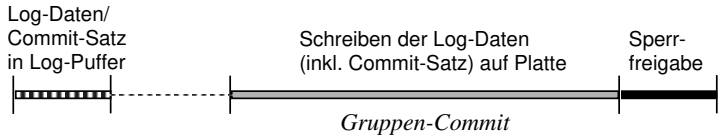
- dynamische Festsetzung des Timer-Wertes durch DBVS wünschenswert

Vergleich verschiedener Commit-Verfahren

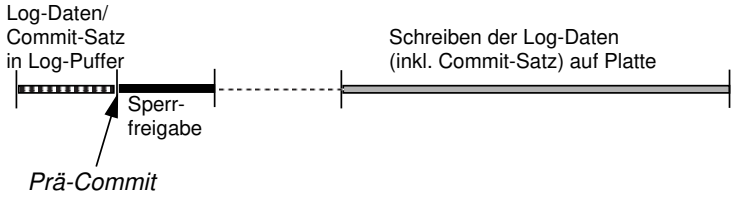
- **Standard-2PC:**



- **Gruppen-Commit:**



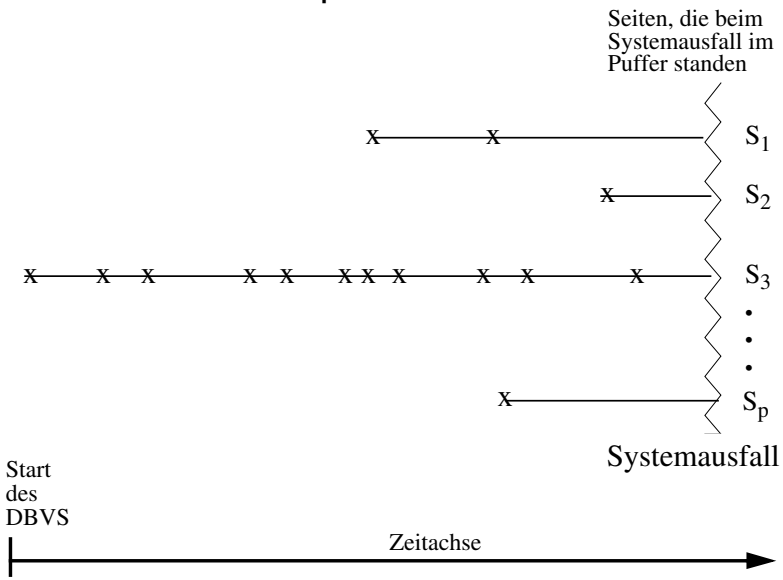
- **Weitere Optimierungsmöglichkeit: *Prä-Commit***
  - Sperren bereits freigeben, wenn Commit-Satz im Log-Puffer steht (vor Schreiben auf Log-Platte)
  - Transaktion kann nur noch durch Systemfehler scheitern
  - In diesem Fall scheitern auch alle ‚abhängigen' Transaktionen, die ungesicherte Änderungen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



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

Sicherungspunkte (*Checkpoints*)

- **Sicherungspunkt: Maßnahme zur Begrenzung des REDO-Aufwandes nach Systemfehlern (NOFORCE)**
- Ohne Sicherungspunkte müßten potentiell alle Änderungen seit Start des DBVS wiederholt werden
- **Besonders kritisch: Hot-Spot-Seiten**



- **Log-Datei**
  - BEGIN_CHKPT-Satz
  - Checkpoint-Informationen
  - END_CHKPT-Satz
- Log-Adresse des letzten Checkpoint-Satzes wird in spezieller Restart-Datei geführt

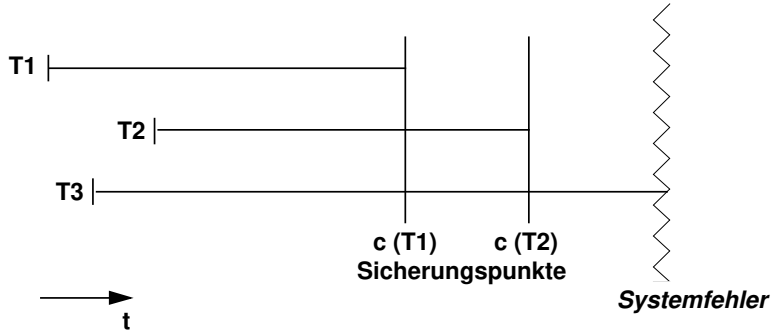
Arten von Sicherungspunkten

- **Direkte Sicherungspunkte**
  - Alle geänderten Seiten im DB-Puffer werden in die permanente DB eingebracht
  - REDO-Recovery beginnt bei letztem Checkpoint
  - 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 Transaktionen, offene Dateien etc.) werden in die Log-Datei geschrieben
  - sehr geringer Checkpoint-Aufwand
  - REDO-Informationen vor letztem Sicherungspunkt sind i. allg. noch zu berücksichtigen
  - Sonderbehandlung von Hot-Spot-Seiten
- **FORCE kann als spezieller Checkpoint-Typ aufgefaßt werden:** nur Seiten einer Transaktion werden ausgeschrieben
  ➥ transaktionsorientierter Sicherungspunkt

Transaktionsorientierte Sicherungspunkte

TOC = Transaction Oriented Checkpoint ≡ FORCE

- **EOT-Behandlung erzwingt das Ausschreiben aller geänderten Seiten der Transaktion aus dem DB-Puffer**
  - Übernahme aller Änderungen in die DB
  - Vermerk in Log-Datei



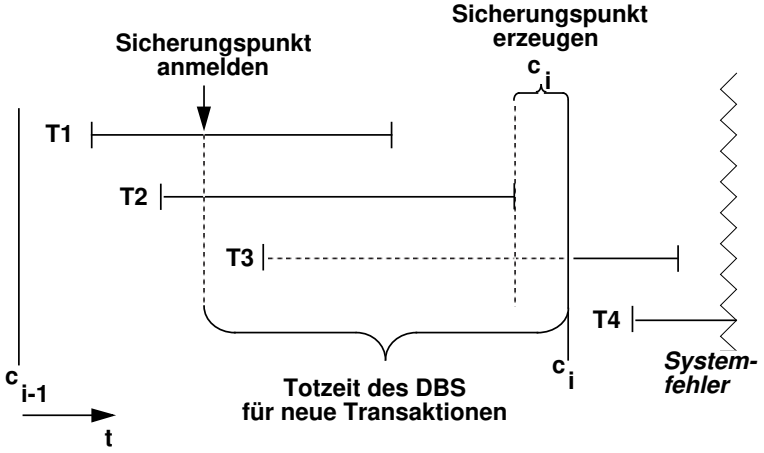
- kein atomares Ausschreiben mehrerer Seiten möglich

  ➥ zumindest bei direktem Einbringen der Seiten UNDO-Recovery vorzusehen (STEAL)

- **Abhängigkeit: NON-ATOMIC, FORCE => STEAL**

Transaktionskonsistente Sicherungspunkte

TCC = Transaction Consistent Checkpoints (logisch konsistent)



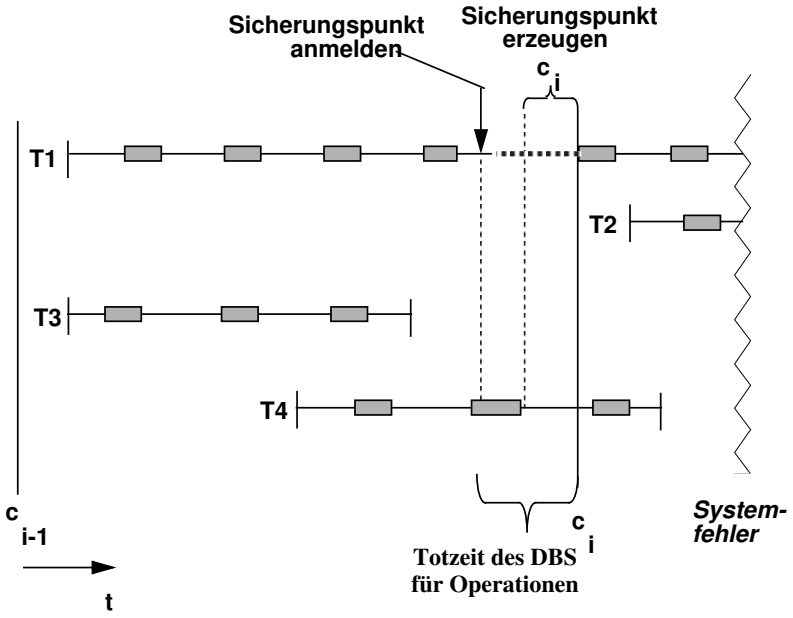
- Ausschreiben ist bis zum Ende aller aktiven Änderungstransaktionen zu verzögern

- Neue Änderungstransaktionen müssen warten, bis Erzeugung des Sicherungspunkts beendet ist

- **Crash-Recovery startet bei letztem Sicherungspunkt**

Aktionskonsistente Sicherungspunkte

ACC = Action Consistent Checkpoints (speicherkonsistent)



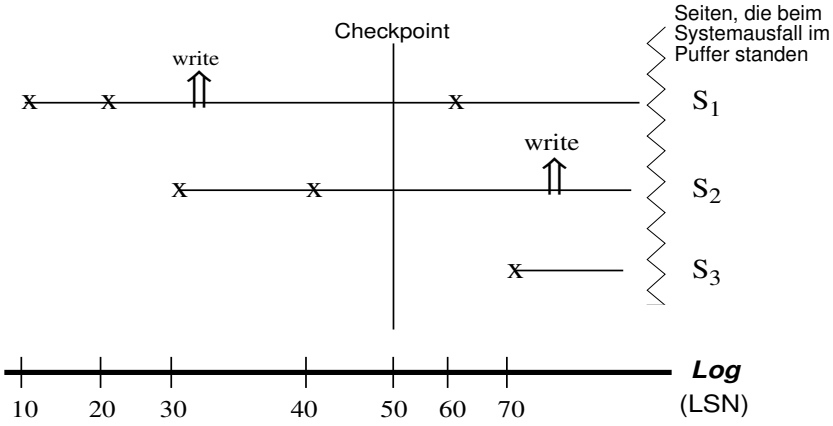
- keine Änderungsanweisungen während des Checkpoints
- geringere Totzeiten als bei TCC,
  dafür Verminderung der Qualität der Sicherungspunkte
- **Crash-Recovery wird nicht durch letzten Sicherungspunkt begrenzt**
- **Abhängigkeit: ACC => STEAL**

Fuzzy Checkpoints

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

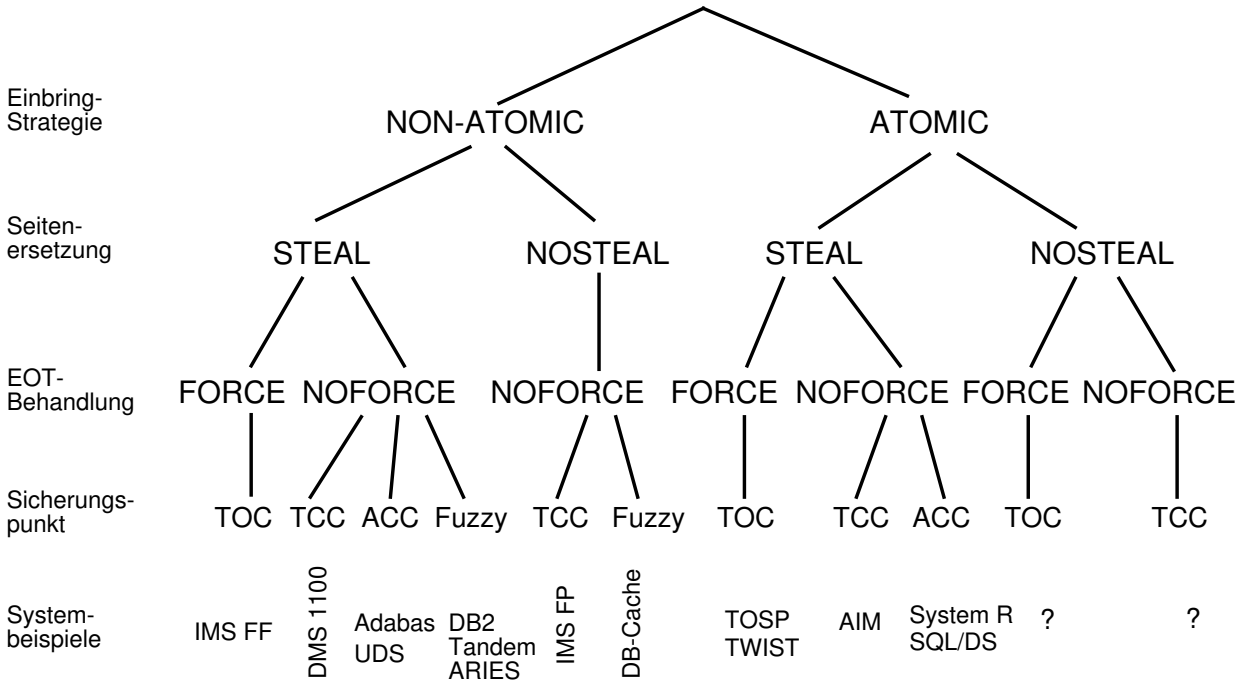
  ➥ nur bei Update-in-Place (NON-ATOMIC) relevant

- **Problem: Bestimmung der Log-Position, an der REDO-Recovery beginnen muß**
  - Pufferverwalter vermerkt sich zu jeder geänderten Seite START-LSN, d. h. Adresse des Log-Satzes der ersten Änderung seit Einlesen von Platte
  - REDO-Recovery nach Crash beginnt bei MinLSN (= MIN(START-LSN))
- Startposition wird in Checkpoint-Information vermerkt
  (daneben laufende Transaktionen, geänderte Seiten, ...)



- **Geänderte Seiten werden asynchron ausgeschrieben**
  - ggf. Kopie der Seite anlegen (für Hot-Spot-Seiten)
  - Seite ausschreiben
  - START-LSN anpassen / zurücksetzen

Klassifikation von DB-Recovery-Verfahren



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

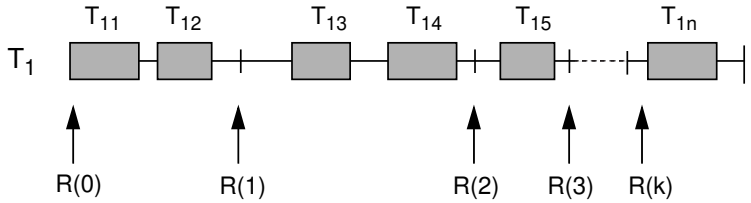
c) Benutze die REDO-Information und wiederhole

d) WAL-Prinzip verhindert diese Situation

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

Zurücksetzen von Transaktionen

- **Transaktions-Recovery**
  - Zurücksetzen einer Transaktion im laufenden DB-Betrieb
  - Nutzung der PrevLSN-Kette im temporären Log
- **Erweiterung zum partiellen Zurücksetzen**
  - Voraussetzung: **transaktionsinterne Rücksetzpunkte** (*Savepoints*)



= atomarer Transaktionsschritt T_{1i}

R(i) = i-ter Rücksetzpunkt

T1 = (T_{11}, T_{12}, ..., T_{1n})

  - Zusätzliche Operationen: SAVE R(i)

    RESTORE R(j)
  - Protokollierung aller Änderungen, Sperren, Cursor-Positionen etc.
  - **UNDO-Operation** bis R(j) in LIFO-Reihenfolge
- **Rücksetzpunkte** müssen vom DBS sowie vom Laufzeitsystem der Programmiersprache unterstützt werden
  - Derzeitige Implementierungen bieten keine Unterstützung von persistenten *Savepoints*!
  - Nach Systemfehler wird Transaktion vollständig zurückgesetzt

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 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
  - zumindest aktionskonsistent

    ➥ DML-Befehle ausführbar (logisches Logging)
  - FORCE: kein REDO
  - NOFORCE:

    a) transaktionskonsistentes Einbringen

    ➥ REDO, jedoch kein UNDO

    b) aktionskonsistentes Einbringen

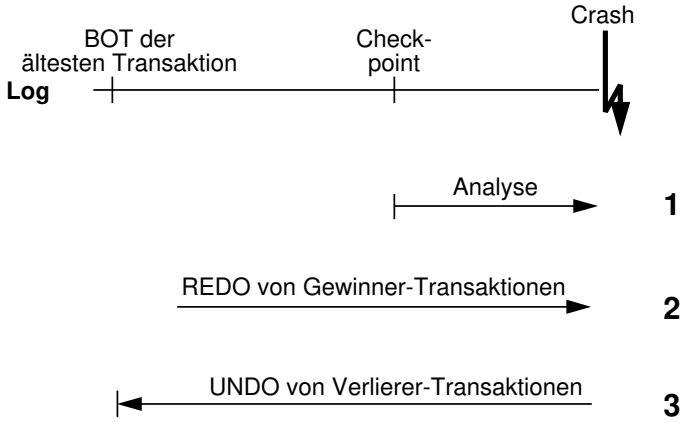
    ➥ UNDO + REDO

Allgemeine Restart-Prozedur

(NON-ATOMIC, STEAL, NOFORCE, CHECKPOINT)

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

1. ***Analyse-Phase*** *(vom letzten Checkpoint bis zum Log-Ende):*
   Bestimmung von ***Gewinner-*** und ***Verlierer-Transaktionen*** sowie der Seiten, die von ihnen geändert wurden
2. ***REDO-Phase:***
   Vorwärtslesen des Log:
   Startpunkt abhängig vom Checkpoint-Typ: MinLSN bei Fuzzy Checkpoint;
   Wiederholung der Änderungen der Gewinner-Transaktionen **(Selektives Redo)** bzw. aller Transaktionen, falls erforderlich (Satzsperren) **(Vollständiges Redo)**
3. ***UNDO-Phase:***
   Rücksetzen der Verlierer-Transaktionen durch Rückwärtslesen des Logs bis zum BOT-Satz der ältesten Verlierer-Transaktion



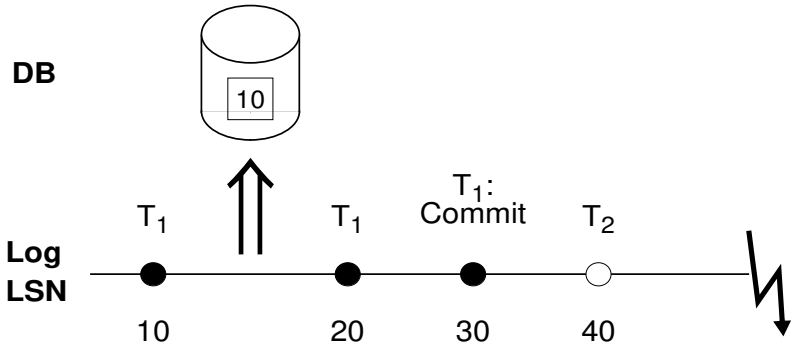
- Für Schritt 2 und 3 sind betroffene DB-Seiten einzulesen
- LSN der Seiten zeigen, ob Log-Informationen anzuwenden sind
- Am Ende sind alle geänderten Seiten wieder auszuschreiben bzw. es wird ein Checkpoint erzeugt

Redo-Recovery

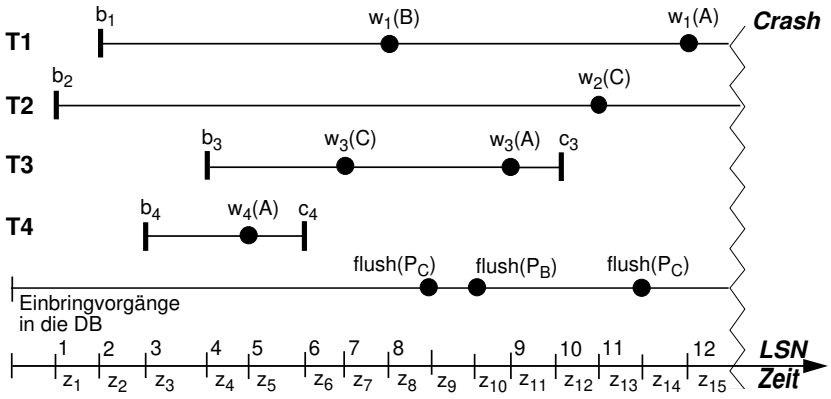
- **Physiologisches und physisches Logging:**
  Notwendigkeit einer Redo-Aktion für Log-Satz wird über PageLSN der betroffenen Seite B angezeigt

```
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
  (Idempotenz der Recovery)



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 0*

Analyse-Phase: Gewinner-Transaktionen: T_3, T_4

Verlierer-Transaktionen: T_1, T_2

Relevante Seiten: P_A, P_B, P_C

Im Restart-Beispiel ändert nie mehr als eine Transaktion gleichzeitig in einer Seite, was einem Einsatz von Seitensperren entspricht. Deshalb ist **Selektives Redo**, also nur das Redo der Gewinner-Transaktionen, 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)

➥ Seiten-LSNs dürfen nicht kleiner werden! Bei erneuten Restart könnten sonst Redo einer Seite mehrfach ausgeführt werden

* „This we know. All things are connected.” (Chief Seattle)

Fehlertoleranz des Restart

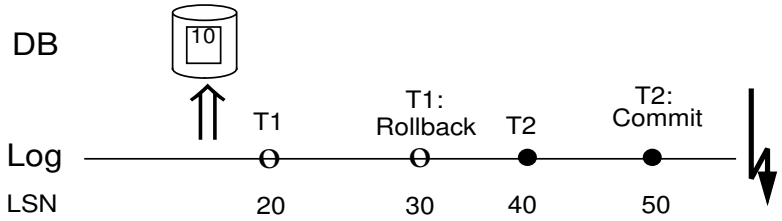
- **Forderung: Fehlertoleranz (Idempotenz) des Restart**

  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 (PageLSN (B) := LSN (L).
    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

- **Bisherige Verwendung der LSN führt zu Problemen**
  in der Undo-Phase bei vorherigem Rollback



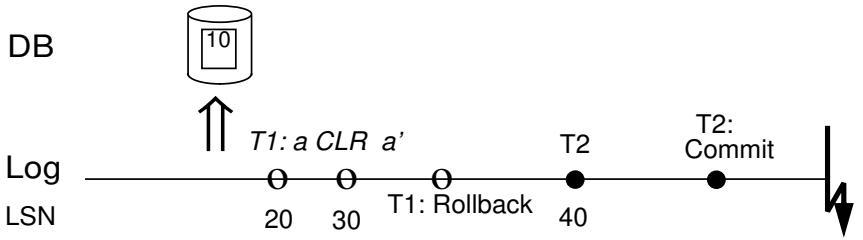
Redo-Phase: Änderung von T2 wird wiederholt (Seiten-LSN := 40)

Undo-Phase: Änderung von T1 wird zurückgesetzt (da 20 < 40), obwohl sie nicht in der Seite vorliegt

➥ Fehler

Compensation Log Records (CLR)

- **Lösung bei selektivem Redo:**
  - Für jede ausgeführte UNDO-Operation wird ein CLR angelegt, der genau wie normale Log-Sätze eine eindeutige LSN zugeteilt bekommt
  - Seiten-LSN wird auch bei UNDO erhöht



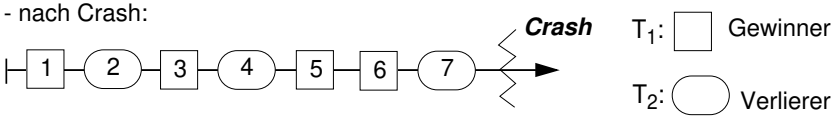
Redo-Phase: Anwendung des CLR (Wiederholen der UNDO-Operation) (Seiten-LSN := 30)
Änderung von T2 wird wiederholt (Seiten-LSN := 40)

Undo-Phase: nur für Transaktionen, die bei Rechnerausfall aktiv waren

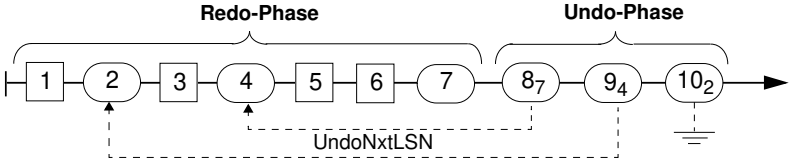
CLRs (2)

- **Allgemeinere Lösung**
  - Bei Satzsperren können mehrere Transaktionen gleichzeitig Änderungen in einer Seite durchführen. Mit einer Seiten-LSN kann jedoch nur die letzte Änderungsoperation kontrolliert werden
  - Einsatz von CLRs bei allen Undo-Operationen: bei Rollback und in der Undo-Phase
  - in der Redo-Phase: **Vollständiges Redo** („repeating history“)
- **Schematische Darstellung der Log-Datei**

- nach Crash:

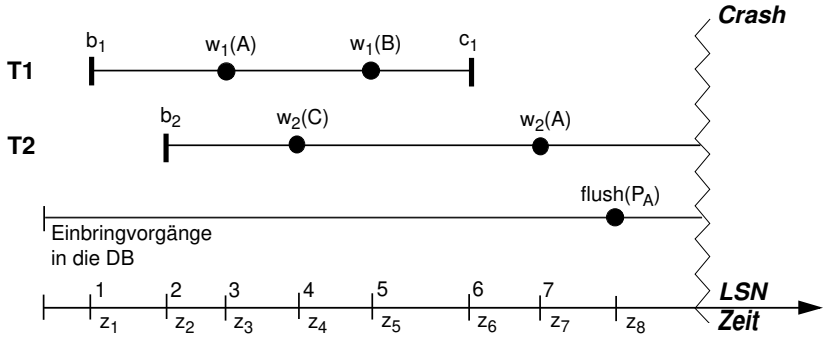


- nach vollständigem Restart:



  - 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)

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 0

Analyse-Phase : Gewinner-Transaktionen: T_1

Verlierer-Transaktionen: T_2

Relevante Seiten: P_A, P_B, P_C

Im Restart-Beispiel 2 wird **Vollständiges Redo** in der Redo-Phase 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 Transaktionen (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)

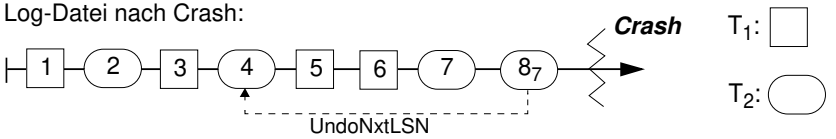
Undo-Phase: Log-Sätze der Verlierer-Transaktion T_2 rückwärts unabhängig von Seiten-LSN prüfen.
Für jeden Log-Satz wird die zugehörige Undo-Operation durchgeführt und mit einem CLR in der Log-Datei vermerkt.

TA	Log-Satz-LSN	Aktion
T_2	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 Transaktionen (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

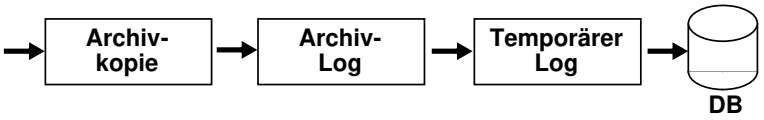
Undo-Phase: Log-Sätze der Verlierer-Transaktion T_2 (inkl. CLRs) rückwärts unabhängig von Seiten-LSN prüfen.
Für jeden Log-Satz wird die zugehörige Undo-Operation durchgeführt und mit einem CLR in der Log-Datei vermerkt.

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

Platten-Recovery

- **Spiegelplatten**
  - schnellste und einfachste Lösung
  - hohe Speicherkosten
  - Doppelfehler nicht auszuschließen
- **Alternative: Archiv-Kopie + Archiv-Log**
- **Archiv-Kopie + Archiv-Log sind längerfristig verfügbar zu halten (auf Band)**

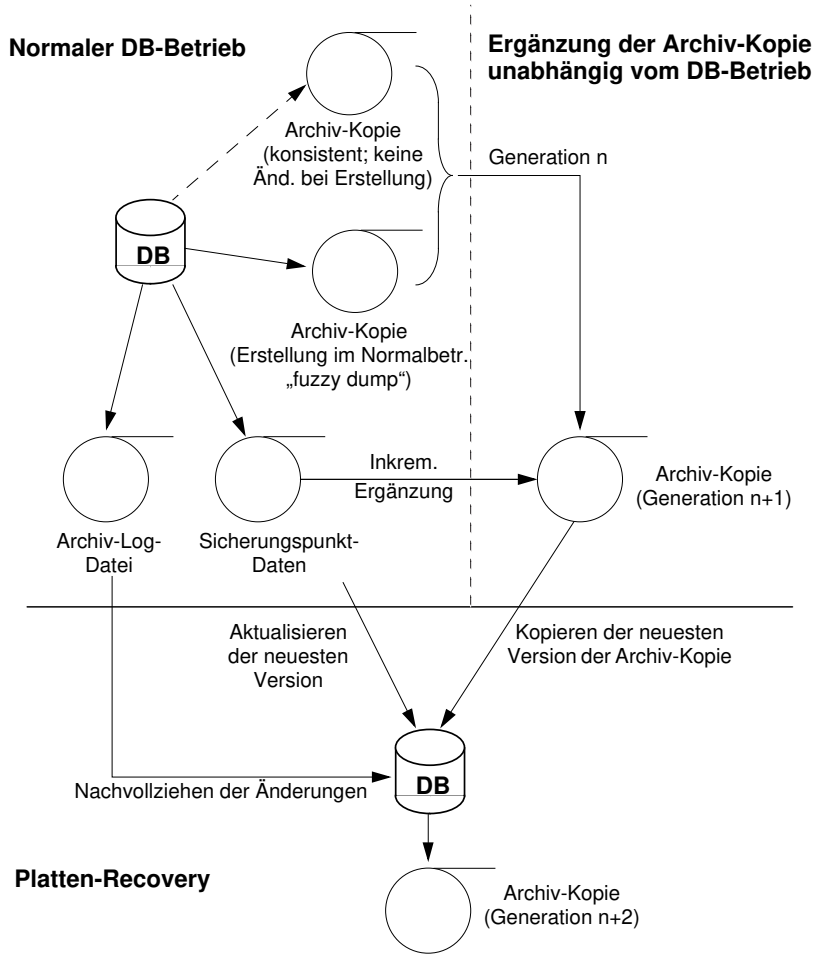
  ➥ Problem von Alterungsfehlern
  - Führen von Generationen der Archiv-Kopie
  - Duplex-Logging für Archiv-Log



- Archiv-Log kann offline aus temporärer Log-Datei abgeleitet werden
- Erstellung von Archivkopien und Archiv-Log erfolgt segmentorientiert

Platten-Recovery - Ein Szenarium

- **Komponenten der Platten-Recovery**



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

Erstellung der Archiv-Kopie

- **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 Plattenfehler Archiv-Log ab Beginn der Dump-Erstellung anzuwenden

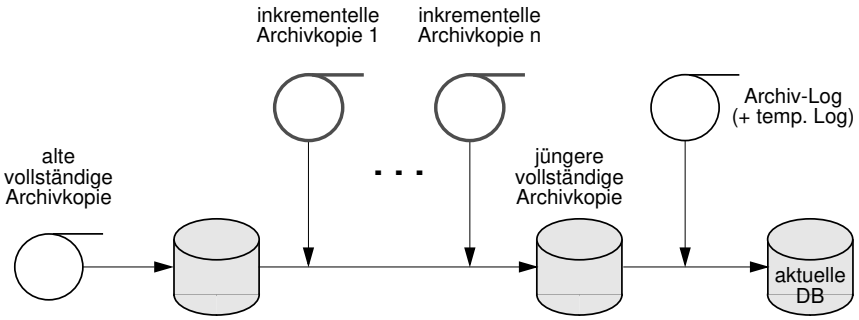
  b2) Aktionskonsistente Archivkopie
  (Voraussetzung bei logischem Operations-Logging)

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

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

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)

Black-/White-Verfahren*

- **Ziel:**
  Erzeugung transaktionskonsistenter Archiv-Kopien

- Spezieller Dumpprozeß zur Erstellung der Archiv-Kopie

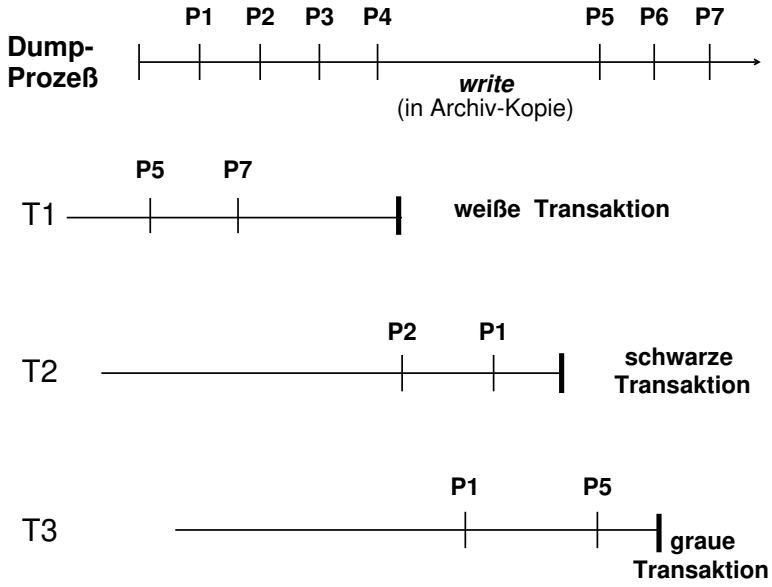
- **Kennzeichnung der Seiten**
  - **Paint-Bit** pro Seite:
    - weiß: Seite wurde noch nicht überprüft
    - schwarz: Seite wurde bereits verarbeitet
  - **Modified-Bit** pro Seite zeigt an, ob eine Änderung seit Erstellung der letzten Archiv-Kopie erfolgte
  - Dumpprozeß färbt alle weißen Seiten schwarz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

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

Black-/White-Verfahren: 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
  - $\Rightarrow$ wird in einigen DBS eingesetzt (DEC RDB)

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
  - Eintrags-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 Checkpoint
  - 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 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:

   Transaktionsbezogene UNDO-Info ist vor Checkpoint 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 Checkpoint-Maßnahmen 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-Operationen 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 geeignetsten