

Skript zur Vorlesung:

Datenbanksysteme I

Wintersemester 2016/2017

Kapitel 10

Transaktionen - Datensicherheit

Vorlesung: Prof. Dr. Christian Böhm

Übungen: Dominik Mautz

<http://www.dbs.ifi.lmu.de/Lehre/DBS>



Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. Synchronisation (Concurrency Control)

Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer

2. Datensicherheit (Recovery)

Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)

3. Integrität (Integrity)

Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer

Fehler- und Recovery-Arten

– Transaktions-Recovery

- **Transaktionsfehler.** Lokaler Fehler einer noch nicht festgeschriebenen TA, z.B. durch
 - Fehler im Anwendungsprogramm
 - Expliziter Abbruch der TA durch den Benutzer (ROLLBACK)
 - Verletzung von Integritätsbedingungen oder Zugriffsrechten
 - Rücksetzung aufgrund von Synchronisationskonflikten
- Behandlung durch **Rücksetzen**
 - *Lokales UNDO*: der ursprüngliche DB-Zustand wie zu BOT wird wiederhergestellt, d.h. Rücksetzen aller Aktionen, die diese TA ausgeführt hat
 - Transaktionsfehler treten relativ häufig auf
 - Behebung innerhalb von Millisekunden notwendig

Fehler- und Recovery-Arten (cont.)

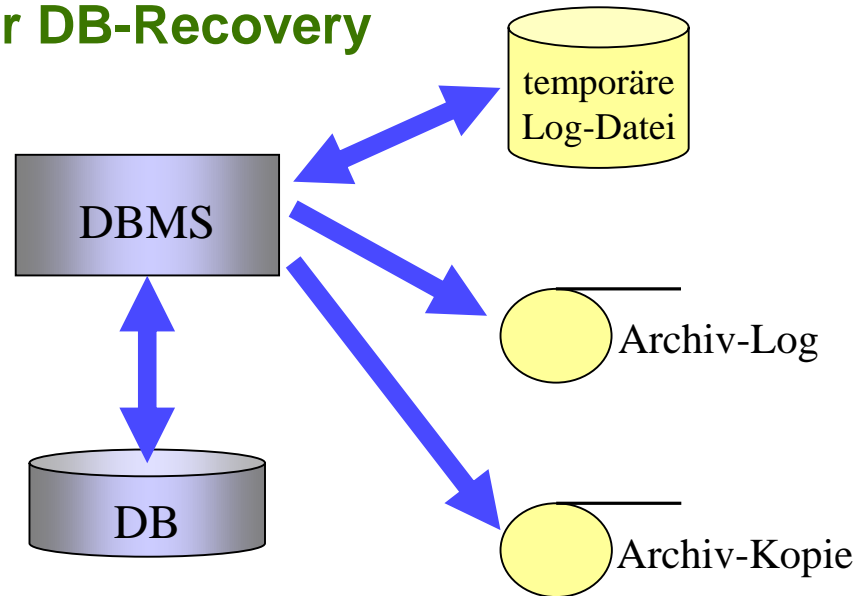
- Crash Recovery
 - **Systemfehler.** Fehler mit Hauptspeicherverlust, d.h. permanente Speicher sind *nicht* betroffen, z.B. durch
 - Stromausfall
 - Ausfall der CPU
 - Absturz des Betriebssystems, ...
 - Behandlung durch **Crash Recovery** (Warmstart)
 - *Globales UNDO*: Rücksetzen aller noch nicht abgeschlossenen TAs, die **bereits** in die DB eingebracht wurden
 - *Globales REDO*: Nachführen aller bereits abgeschlossenen TAs, die **noch nicht** in die DB eingebracht wurden
 - Systemfehler treten i.d.R. im Intervall von Tagen auf
→ Recoverydauer einige Minuten

Fehler- und Recovery-Arten (cont.)

– Geräte-Recovery

- **Medienfehler.** Fehler mit Hintergrundspeicherverlust, d.h. Verlust von permanenten Daten, z.B. durch
 - Plattencrash
 - Brand, Wasserschaden, ...
 - Fehler in Systemprogrammen, die zu einem Datenverlust führen
- Behandlung durch **Geräte-Recovery** (Kaltstart)
 - Aufsetzen auf einem früheren, gesicherten DB-Zustand (Archivkopie)
 - *Globales REDO*: Nachführen aller TAs, die nach dem Erzeugen der Sicherheitskopie abgeschlossen wurden
 - Medienfehler treten eher selten auf (mehrere Jahre)
 - Recoverydauer einige Stunden / Tage
 - **Wichtig**: regelmäßige Sicherungskopien der DB notwendig

Systemkomponenten der DB-Recovery



- Behandlung von Transaktions- und Systemfehlern

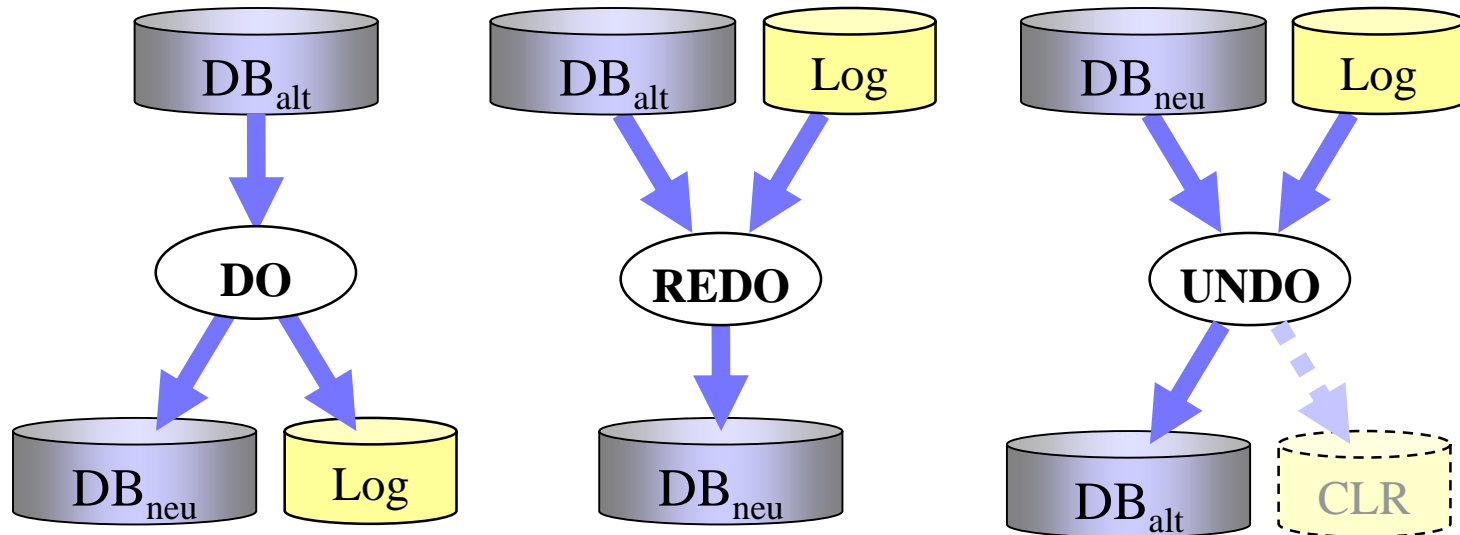
DB + temporäre Log-Datei → DB

- Behandlung von Medienfehlern

Archiv-Kopie + Archiv-Log → DB

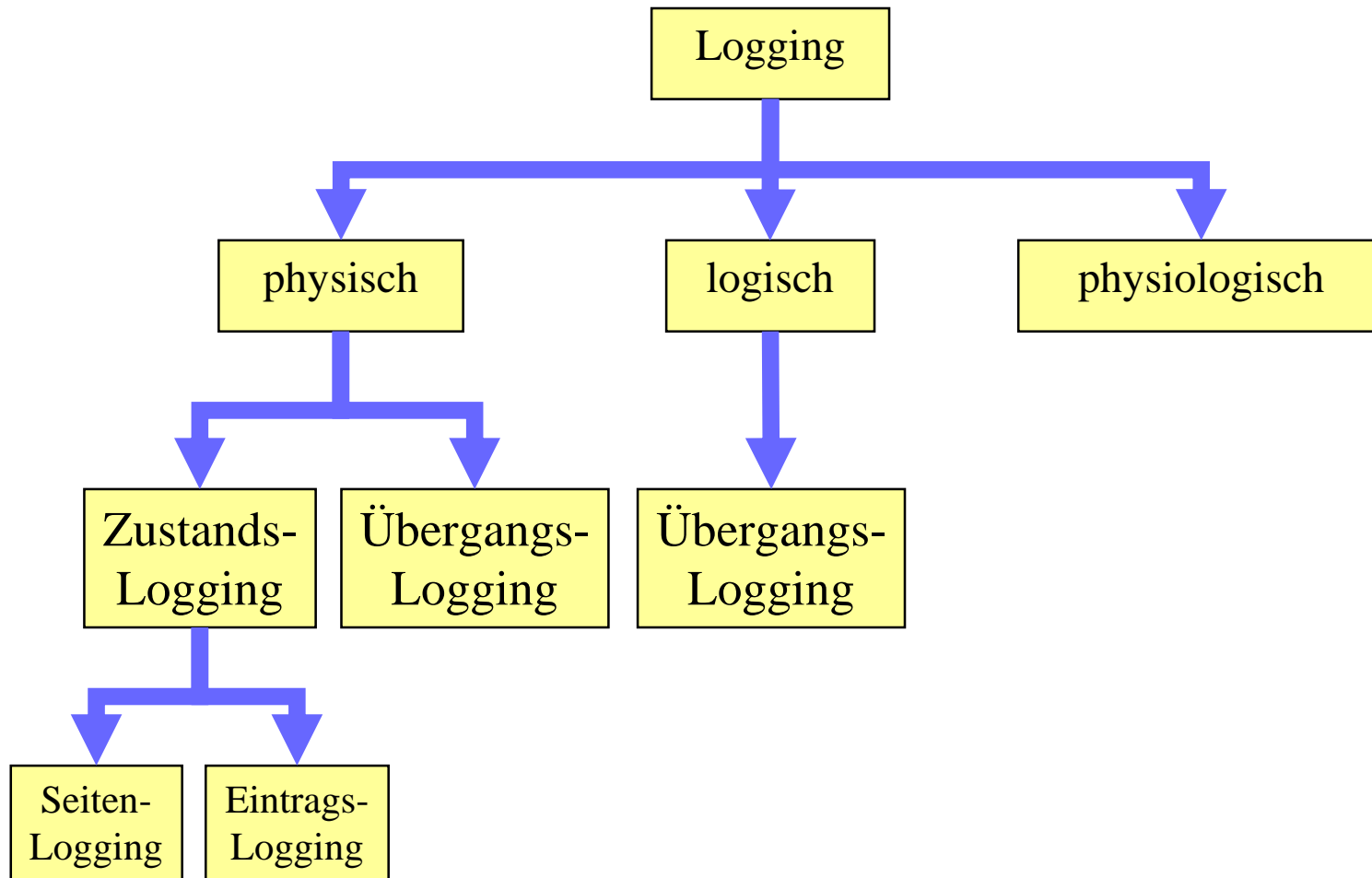
Aufgaben des Logging

- Für jede Änderungsoperation auf der Datenbank im Normalbetrieb (**DO**) benötigt man Protokolleinträge für
 - **REDO**: Information zum Nachvollziehen der Änderungen erfolgreicher TAs
 - **UNDO**: Information zum Zurücknehmen der Änderungen unvollständiger TAs



CLR = Compensation Log Record (zur Behandlung von Fehlern während der Recovery)

Klassifikation von Logging-Verfahren



Physisches Logging

- Protokoll auf der Ebene der physischen Objekte (Seiten, Datensätze, Indexeinträge)
- Zustandslogging
 - Protokollierung der Werte vor und nach jeder Änderung: Alte Zustände *BFIM* (Before-Images) und neue Zustände *AFIM* (After-Images) der geänderten Objekte werden in die Log-Datei geschrieben
- Übergangslgging
 - Protokollierung der Zustandsdifferenz zwischen BFIM und AFIM

- Zustandslogging auf Seitenebene
 - vollständige Kopien von Seiten werden protokolliert
 - Recovery sehr einfach und schnell (Seiten einfach zurückkopieren)
 - sehr großer Logumfang und hohe I/O-Kosten auch bei nur kleinen Änderungen
 - Seitenlogging impliziert Seitensperren → hohe Konfliktrate bei Synchronisation
- Zustandslogging auf Eintragsebene
 - statt ganzer Seiten werden nur tatsächlich geänderte Einträge protokolliert
 - kleinere Sperrgranulate als Seiten möglich
 - Protokollgröße reduziert sich typischerweise um mind. 1 Größenordnung
 - Log-Einträge werden in Puffer gesammelt → wesentlich weniger Plattenzugriffe
 - Recovery ist aufwändiger: zu ändernde Datenbankseiten müssen vollständig in den Hauptspeicher geladen werden, um die Log-Einträge anwenden zu können

- Übergangslogging
 - Protokollierung der Zustandsdifferenz zwischen *BFIM* und *AFIM*
 - Aus *BFIM* muss *AFIM* berechenbar sein (u.u.)
 - Realisierbar durch XOR-Operation \oplus (eXclusive-OR)¹:

	Zustands-Logging	Übergangs-Logging
DO Änderung $A_{alt} \rightarrow A_{neu}$	Protokollierung von $BFIM = A_{alt}, AFIM = A_{neu}$	Protokollierung von $D = A_{alt} \oplus A_{neu}$
REDO (in DB liegt A_{alt})	Überschreibe A_{alt} mit <i>AFIM</i>	$A_{neu} = A_{alt} \oplus D$
UNDO (in DB liegt A_{neu})	Überschreibe A_{neu} mit <i>BFIM</i>	$A_{alt} = A_{neu} \oplus D$

¹ XOR-Operation:

XOR:

$0 \oplus 0 = 0$

$0 \oplus 1 = 1$

$1 \oplus 0 = 1$

$1 \oplus 1 = 0$

Logisches Logging

- Spezielle Form des Übergangs-Logging: Protokollierung von Änderungsoperationen mit ihren aktuellen Parametern
- Protokoll auf hoher Abstraktionsebene ermöglicht kurze Log-Einträge
- Probleme für **REDO**: Änderungen umfassen typischerweise mehrere Seiten (Tabelle, Indexe)
 - Atomares Einbringen der Mehrfachänderungen schwierig
 - Logische Änderungen sind aufwändiger durchzuführen als physische Änderungen
- Probleme für **UNDO**: Mengenorientierte Änderungen können sehr aufwändige Protokolleinträge verursachen:
 - Bsp.: `DELETE FROM Products WHERE Group = 'G1'`
=> **UNDO** erfordert viele Einfügungen, falls Produktgruppe G1 umfangreich ist
 - Bsp.: `UPDATE Products SET Group = 'G2' WHERE Group = 'G1'`
=> **UNDO** muss alte und neue Produkte der Gruppe G2 unterscheiden

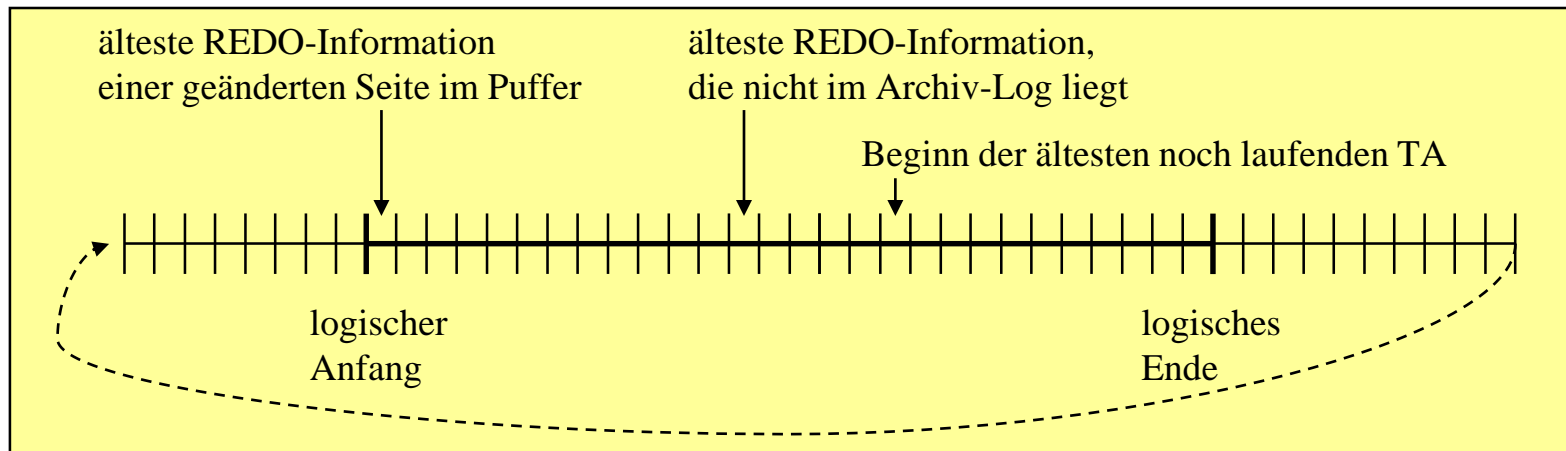
Physiologisches Logging

- Kombination von physischem und logischem Logging:
Protokollierung von elementaren Operationen innerhalb einer Seite
 - Physical-to-a-page
 - Protokollierungseinheiten sind geänderte Seiten
 - gut verträglich mit Pufferverwaltung und direktem (atomarem) Einbringen
 - Logical-within-a-page
 - logische Protokollierung der Änderungen auf einer Seite
- Bewertung
 - Log-Einträge beziehen sich nicht auf mehrere Seiten wie bei logischem Logging
 - Dadurch einfachere Recovery als bei logischem Logging
 - Log-Datei ist länger als bei logischem Logging aber kürzer als bei physischem Logging
 - Flexibler als physisches Logging wegen variabler Objektpositionen auf Seiten.

Die Log-Datei

- Art der Protokolleinträge
 - Beginn, Commit und Rollback von Transaktionen
 - Änderungen des DB-Zustandes durch Transaktionen
 - Sicherungspunkte (Checkpoints)
- Komponenten von Änderungseinträgen: (LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN)
 - LSN (Log Sequence Number): eindeutige Kennung des Log-Eintrags in chronologischer Reihenfolge
 - TA-Id: eindeutige Kennung der TA, die die Änderung durchgeführt hat
 - Page-Id : Kennung der Seite auf der die Änderungsoperation vollzogen wurde (ein Eintrag pro geänderter Seite)
 - REDO: gibt an, wie die Änderung nachvollzogen werden kann
 - UNDO: beschreibt, wie die Änderung rückgängig gemacht werden kann
 - PrevLSN: Zeiger auf vorhergehenden Log-Eintrag der jeweiligen TA (Effizienzgründe)

- Die Log-Datei ist eine **sequentielle** Datei: Schreiben neuer Protokolldaten an das aktuelle Dateiende
=> Ringpuffer-Organisation
- Log-Daten sind für **Crash-Recovery** nur begrenzte Zeit relevant:
 - **UNDO**-Sätze für erfolgreich beendete TAs werden nicht mehr benötigt
 - Nach Einbringen der Seite in die DB wird REDO-Information nicht mehr benötigt



- **REDO-Information für Geräte-Recovery** ist im Archiv-Log zu sammeln

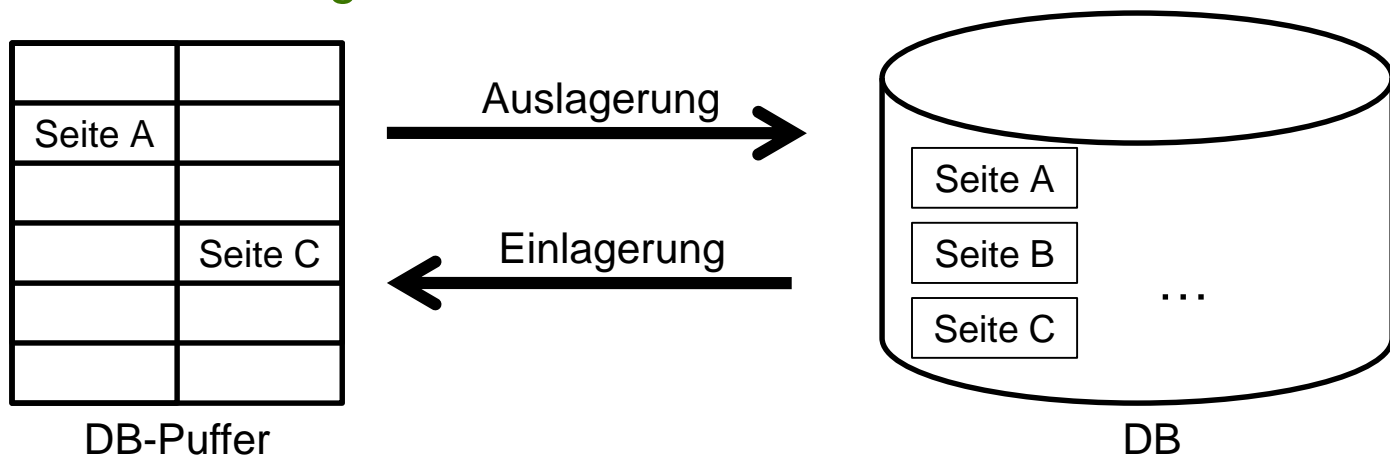
- Beispiel

Ablauf T_1	Ablauf T_2	Log-Eintrag <i>(LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN)</i>
begin read(A, a_1) $a_1 := a_1 - 50$ write (A, a_1) read(B, b_1) //70 $b_1 := 50$ write (B, b_1) commit	begin read(C, c_2) //80 $c_2 := 100$ write (C, c_2) read(A, a_2) $a_2 := a_2 - 100$ write (A, a_2) commit	(#1, T_1 , begin, 0) (#2, T_2 , begin, 0) (#3, T_1 , p_A , $A-=50, A+=50, \#1$) (#4, T_2 , p_C , $C=100, C=80, \#2$) (#5, T_1 , p_B , $B=50, B=70, \#3$) (#6, T_1 , commit, #5) (#7, T_2 , p_A , $A-=100, A+=100, \#4$) (#8, T_2 , commit, #7)

(hier: logisches Logging)

Die Speicherhierarchie

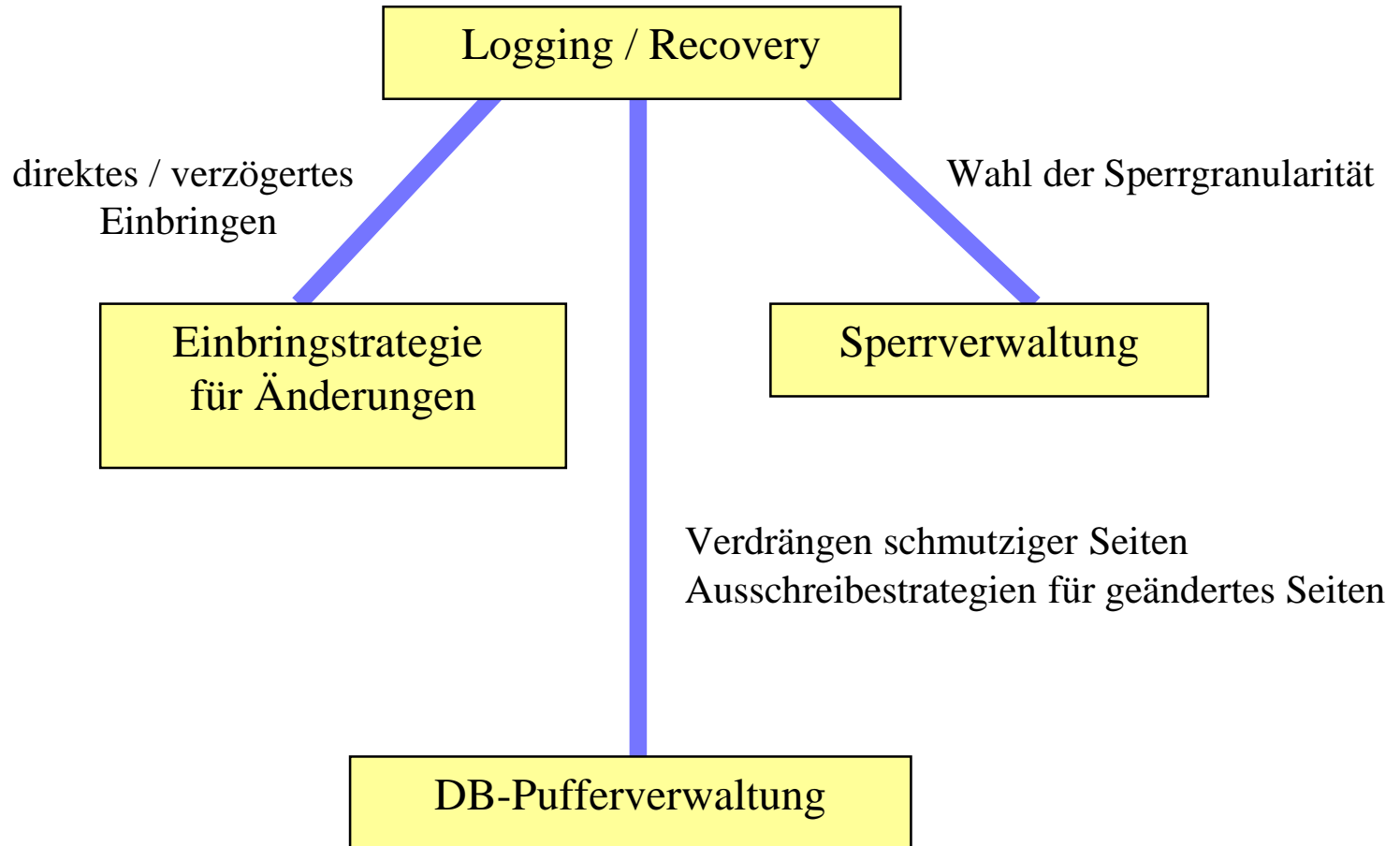
- i.d.R. besteht die Speicherhierarchie bei DBMS aus zwei Ebenen
 - DBMS-Puffer (Hauptspeicher) [kurz: DB-Puffer]
 - DB (Hintegrundspeicher)
- Im laufenden Betrieb werden die Operationen der einzelnen TAs im DB-Puffer ausgeführt
- Die DB muss gemäß dem ACID-Prinzip transaktionskonsistent gehalten werden, d.h. Änderungen durch TAs müssen nach dem Commit in die DB eingebracht werden



Abhängigkeiten

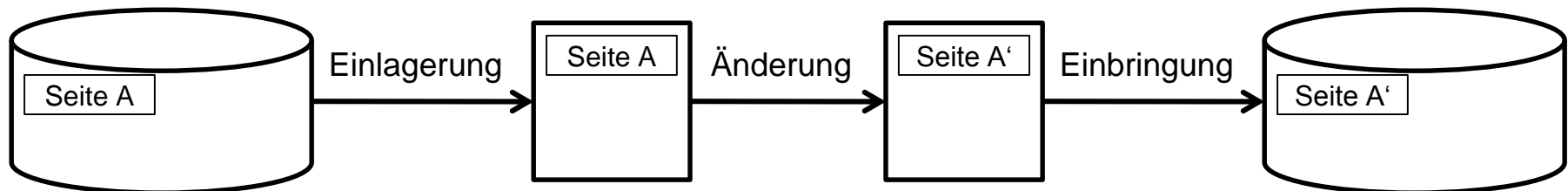
- Aus der zweistufigen Speicherhierarchie ergeben sich insbesondere Abhängigkeiten der Recovery/des Loggings zur Speicherverwaltung
 - DB-Puffer ist begrenzt => was passiert, wenn Puffer voll?
=> Pufferverwaltung (**Verdrängungsstrategien**)
 - Wann schreibe ich Änderungen in die Datenbank?
=> Pufferverwaltung (**Ausschreibestrategien**)
 - Wie schreibe ich die Änderungen aus?
=> HGS-Verwaltung (**Einbringungsstrategien**)
- => Abhängig davon: wann schreibe ich Log-Datei auf Platte?
- Wie wir sehen werden, besteht zudem eine Abhängigkeit der Recovery/des Loggings zur Sperrverwaltung (bei pessimistischer Synchronisation)

Schematischer Überblick der Abhängigkeiten



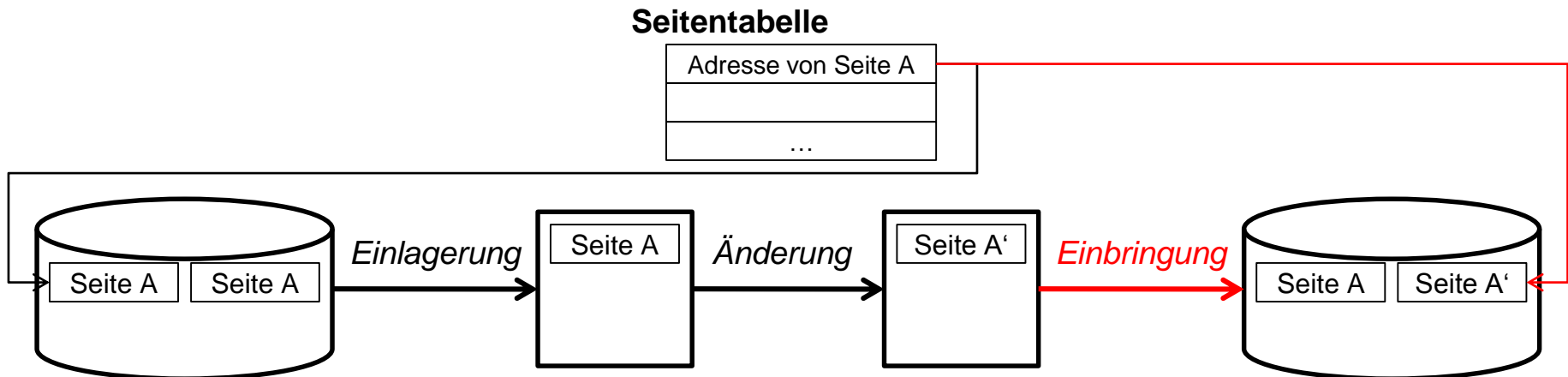
Einbringungstrategie: Direktes Einbringen (NonAtomic, Update-in-Place)

- Jede Seite hat eine Speicheradresse auf der Platte
- Geänderte Seiten werden immer auf ihren Block zurück geschrieben, d.h. der alte Inhalt der Seite in der DB wird dabei überschrieben
- Ausschreiben einer Seite ist dadurch gleichzeitig Einbringen in die permanente DB (es gibt keinen Zwischenspeicher für Seiten)
- Es ist nicht möglich mehrere Seiten atomar einzubringen, d.h. Unterbrechungsfreiheit des Einbringens kann nicht garantiert werden (daher: **NonAtomic**)
- Dies ist die gängigste Methode in heutigen DBMS
- **UNDO**-Informationen müssen explizit gespeichert werden



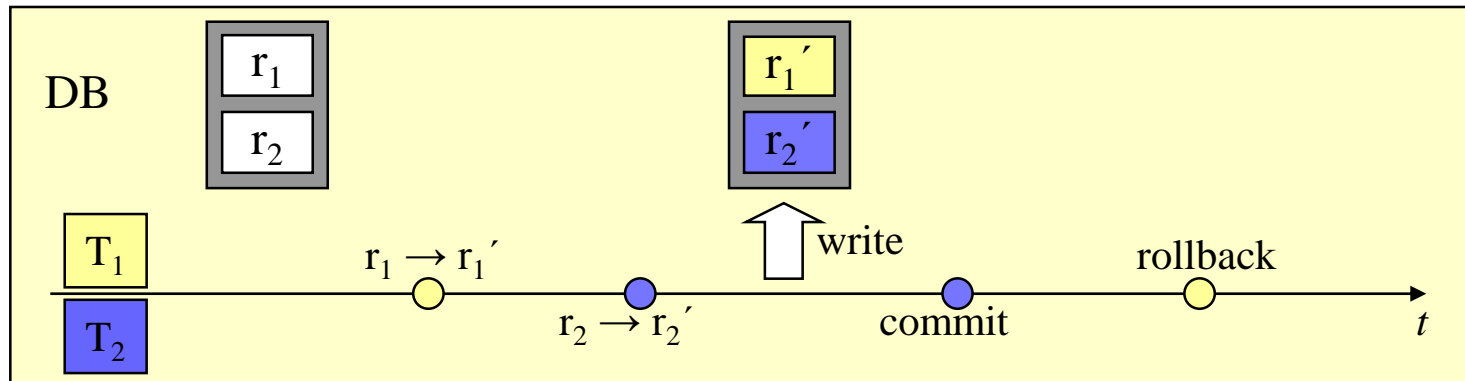
Einbringungstrategie: Indirektes Einbringen (Atomic)

- Geänderte Seite wird in separaten Block auf Platte geschrieben
 - Twin-Block-Verfahren: jede Seite hat zwei Blöcke auf der Platte
 - Schattenspeichertechnik: nur modifizierte Seiten haben zwei Blöcke
- Atomares Einbringen mehrerer geänderter Seiten ist durch Umschalten von Seitentabellen möglich (daher: **Atomic**)
- Alte Versionen der Objekte bleiben erhalten, d.h. es muss keine **UNDO**-Information explizit gespeichert werden



Einfluss der Sperrgranularität

- Log-Granularität muss kleiner oder gleich der Sperrgranularität sein, sonst Lost Updates möglich
- D.h. Satzsperrern erzwingen feine Log-Granulate
- Beispiel für Problem bei “Satzsperrern mit Seitenlogging”



- T_1 , T_2 ändern die Datensätze r_1 , r_2 , die auf derselben DB-Seite liegen
- Die Seite wird in die DB zurück geschrieben, T_2 endet mit COMMIT
- Falls T_1 zurückgesetzt wird, geht auch die Änderung $r_2 \rightarrow r_2'$ verloren
- Lost Update, d.h. Verstoß gegen die Dauerhaftigkeit des COMMIT

Pufferverwaltung: Verdrängungsstrategien

- Ersetzung schmutziger Seiten im Puffer (Wohin werden Seiten verdrängt? Warum nur schmutzige Seiten?)

Seite ist schmutzig wenn: $\text{SeitePuffer} \neq \text{SeiteDB}$

- **No-Steal**
 - Schmutzige Seiten dürfen nicht aus dem Puffer entfernt werden
 - DB enthält keine Änderungen nicht-erfolgreicher TAs
 - **UNDO-Recovery** ist nicht erforderlich
 - Probleme bei langen Änderungs-TAs, da große Teile des Puffers blockiert werden => Einschränkung der Parallelität
- **Steal**
 - Schmutzige Seiten dürfen jederzeit ersetzt und in die DB eingebracht werden
 - DB kann unbestätigte Änderungen enthalten
 - **UNDO-Recovery** ist erforderlich
 - effektivere Puffernutzung bei langen TAs mit vielen Änderungen

Pufferverwaltung: Ausschreibestrategien (EOT-Behandlung)

- Wann werden Änderungen in die DB eingebracht?
 - **Force**
 - Alle geänderte Seiten werden spätestens bei EOT (vor COMMIT) in die DB geschrieben
 - keine **REDO**-Recovery erforderlich bei Systemfehler
 - hoher I/O-Aufwand, da Änderungen jeder TA einzeln geschrieben werden
 - Vielzahl an Schreibvorgängen führt zu schlechteren Antwortzeiten, länger gehaltenen Sperrungen und damit zu mehr Sperrkonflikten
 - Große DB-Puffer werden schlecht genutzt
 - **No-Force**
 - Änderungen können auch erst nach dem COMMIT in die DB geschrieben werden
 - Beim COMMIT werden lediglich **REDO**-Informationen in die Log-Datei geschrieben
 - **REDO**-Recovery erforderlich bei Systemfehler
 - Änderungen auf einer Seite von mehreren TAs können gesammelt werden

Kombination:

	No-Steal	Steal
Force	kein <i>UNDO</i> – kein <i>REDO</i> (nicht für Update-in-Place)	<i>UNDO</i> – kein <i>REDO</i>
No-Force	kein <i>UNDO</i> – <i>REDO</i>	<i>UNDO</i> – <i>REDO</i>

- Bewertung **Steal / No-Force**
 - erfordert zwar **UNDO** als auch **REDO**, ist aber allgemeinste Lösung
 - beste Leistungsmerkmale im Normalbetrieb
- Bewertung **No-Steal / Force**
 - optimiert den Fehlerfall auf Kosten des Normalfalls (sehr teures COMMIT)
 - für *Update-in-Place* nicht durchführbar:
 - wegen **No-Steal** dürfen Änderungen erst nach COMMIT in die DB gelangen, was jedoch **Force** widerspricht (**No-Steal** → **No-Force**)
 - wegen Force müssten Änderungen vor dem COMMIT in der DB stehen, was bei *Update-in-Place* unterbrochen werden kann, **UNDO** wäre nötig (**Force** → **Steal**)

WAL-Prinzip und COMMIT-Regel

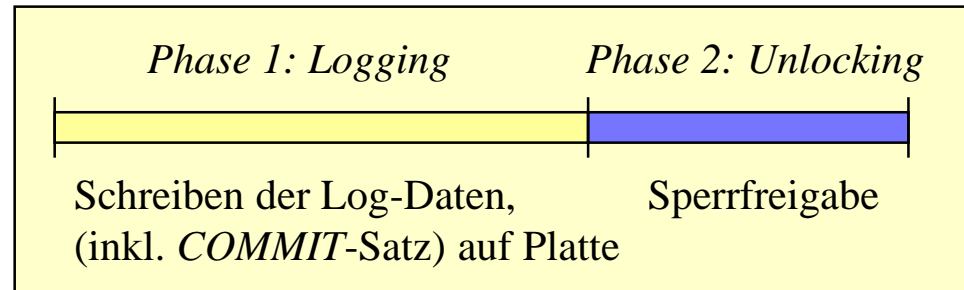
- WAL-Prinzip (**Write-Ahead-Log**)
 - **UNDO**-Information (z.B. BFIM) muss vor Änderung der DB im Protokoll stehen
 - Wichtig, um schmutzige Änderungen rückgängig zu machen
 - Nur relevant für *Steal*
 - Wichtig bei direktem Einbringen
- COMMIT-Regel (**Force-Log-at-Commit**)
 - **REDO**-Information (z.B. AFIM) muss vor dem COMMIT im Protokoll stehen
 - Voraussetzung für Crash-Recovery bei *No-Force*
 - Erforderlich für Geräte-Recovery (auch bei *Force*)
 - Gilt für direkte und indirekte Einbringstrategien gleichermaßen
- Bemerkung: Um die chronologische Reihenfolge im Ringpuffer zu wahren, werden alle Log-Einträge bis zum letzten notwendigen ausgeschrieben, d.h. es werden keine Log-Einträge übergangen

COMMIT-Verarbeitung

- **Standard Zwei-Phasen-Commit**

- *Phase 1: Logging*

- Überprüfen der verzögerten Integritätsbedingungen
- Logging der **REDO**-Informationen incl. COMMIT-Satzes



- *Phase 2: Unlocking*

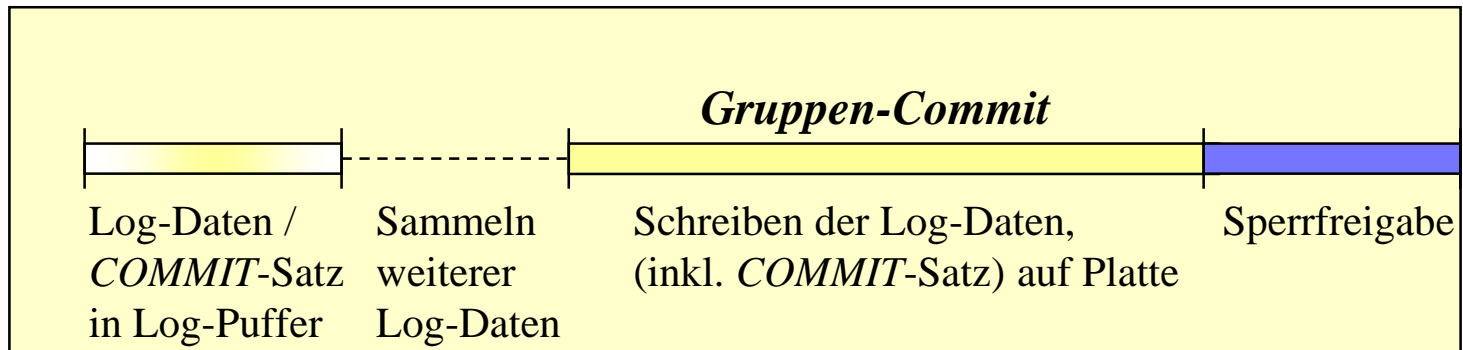
- Freigabe der Sperren (Sichtbarmachen der Änderungen)
- Bestätigung des COMMIT an das Anwendungsprogramm

- Problem

- COMMIT-Regel verlangt Ausschreiben des Log-Puffers bei jedem COMMIT
- Beeinträchtigung für kurze TAs, deren Log-Daten weniger als eine Seite umfassen
- Durchsatz an TAs ist eingeschränkt

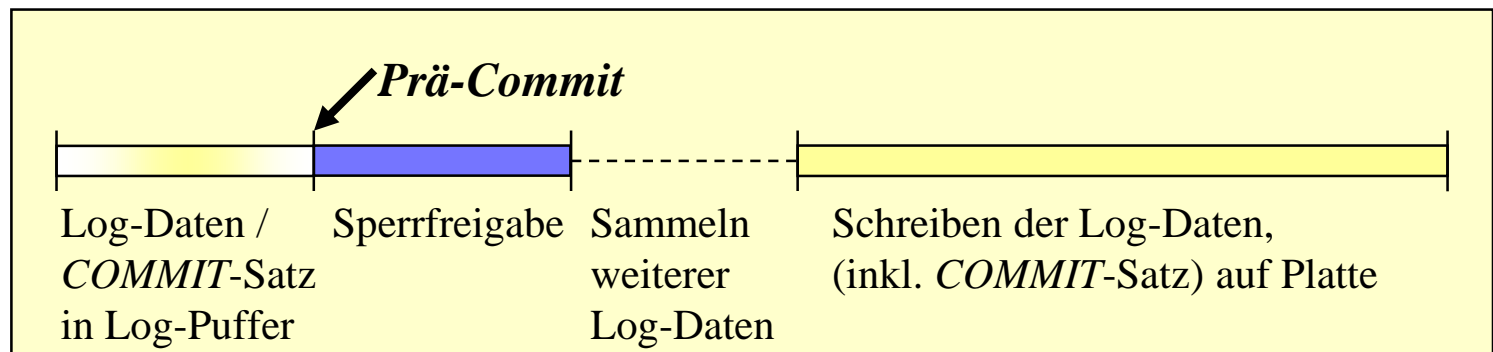
- **Gruppen-Commit**

- Log-Daten mehrerer TAs werden im Puffer gesammelt
- Log-Puffer wird auf Platte geschrieben, sobald Puffer gefüllt ist oder nach Timeout
- Vorteil: Reduktion der Plattenzugriffe und höhere Transaktions-raten möglich
- Nachteil: längere Sperrdauer führt zu längeren Antwortzeiten
- In der Praxis: wird von zahlreichen DBS unterstützt

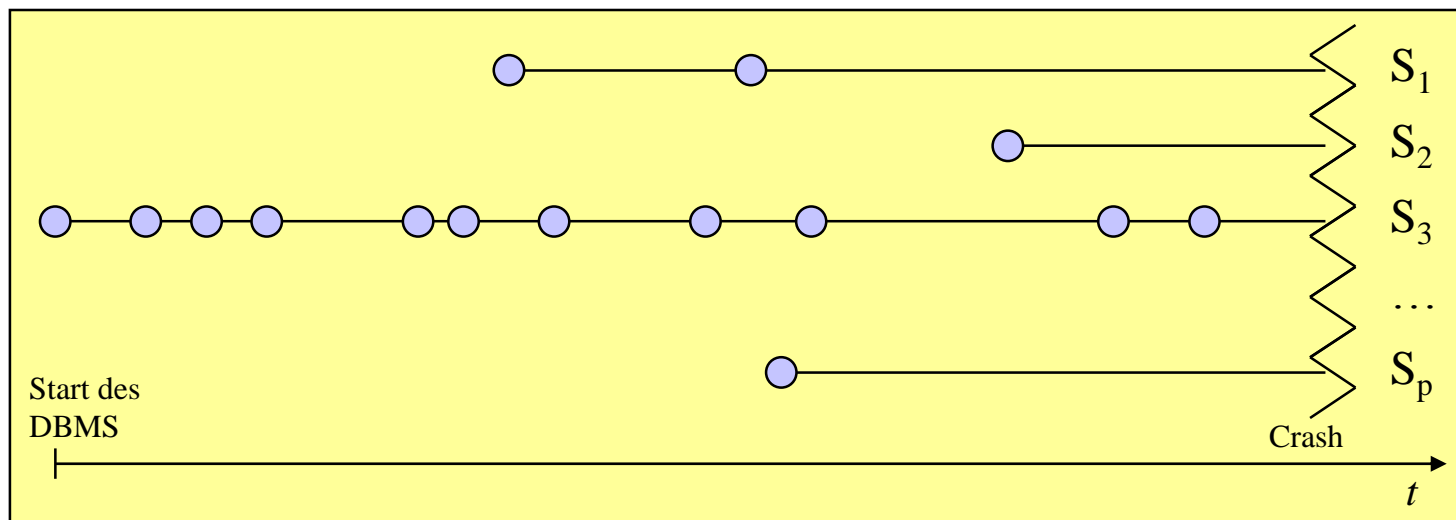


- **Prä-Commit**

- Vermeidung der langen Sperrzeiten des Gruppen-Commit indem Sperren bereits freigegeben werden, wenn COMMIT-Satz im Log-Puffer steht
- Ist Prä-Commit zulässig?
- Normalfall: ändernde TA kommt erfolgreich zu Ende, Änderungen sind gültig
- Fehlerfall: Abbruch der TA nur noch durch Systemfehler möglich; bei Systemfehler werden auch die anderen laufenden TAs abgebrochen, "schmutziges Lesen" kann sich also nicht auf DB auswirken



- Sicherungspunkte sind eine Maßnahme zur Begrenzung des **REDO-**Aufwands nach Systemfehlern
- Ohne Sicherungspunkte müssten potentiell alle Änderungen seit Start des DBMS wiederholt werden
- Besonders kritisch: Hot-Spot-Seiten, die (fast) nie aus dem Puffer verdrängt werden



- Durchführung von Sicherungspunkten
 - Spezielle Log-Einträge:
 - BEGIN_CHKPT
 - Info über laufende TAs
 - END_CHKPT
 - *LSN* des letzten vollständig ausgeführten Sicherungspunktes wird in Restart-Datei geführt
- Häufigkeit von Sicherungspunkten
 - **zu selten**: hoher **REDO**-Aufwand
 - **zu oft**: hoher Overhead im Normalbetrieb
 - z.B. Sicherungspunkte nach bestimmter Anzahl von Log-Sätzen einfügen

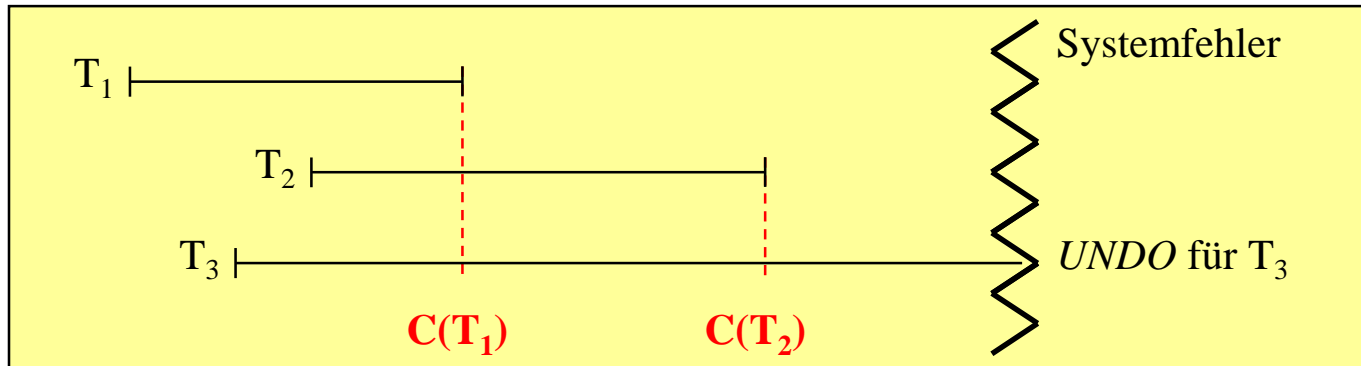
Direkte Sicherungspunkte

- Charakterisierung
 - Alle geänderten Seiten werden am Sicherungspunkt in die persistente DB (Platte) geschrieben
 - Zeitbedarf steigt mit dem zeitlichen Abstand der Sicherungspunkte
 - Multi-Page-Access hilft, Schreibkopf-Positionierungen zu minimieren
 - **REDO**-Recovery kann beim letzten vollständig ausgeführten Checkpoint beginnen
- 3 Arten
 - Transaktions-orientierte Sicherungspunkte (**TOC**)
 - Transaktions-konsistente Sicherungspunkte (**TCC**)
 - Aktions-konsistente Sicherungspunkte (**ACC**)

TOC: TA-orientierte Sicherungspunkte

- TOC entspricht *Force*, d.h. Ausschreiben aller Änderungen beim COMMIT oder anders gesagt: jedes COMMIT definiert einen Sicherungspunkt
- Nicht alle Seiten im Puffer werden geschrieben, sondern nur Änderungen der jeweiligen TA
- Sicherungspunkt bezieht sich immer auf genau eine TA (die den Sicherungspunkt mit COMMIT ausgelöst hat)
- **UNDO-Recovery**
Bei *Update-in-Place* ist **UNDO** nötig (*Force* → *Steal*), **UNDO** beginnt dann beim letzten Sicherungspunkt
- **REDO-Recovery** garnicht nötig (*Force*!)

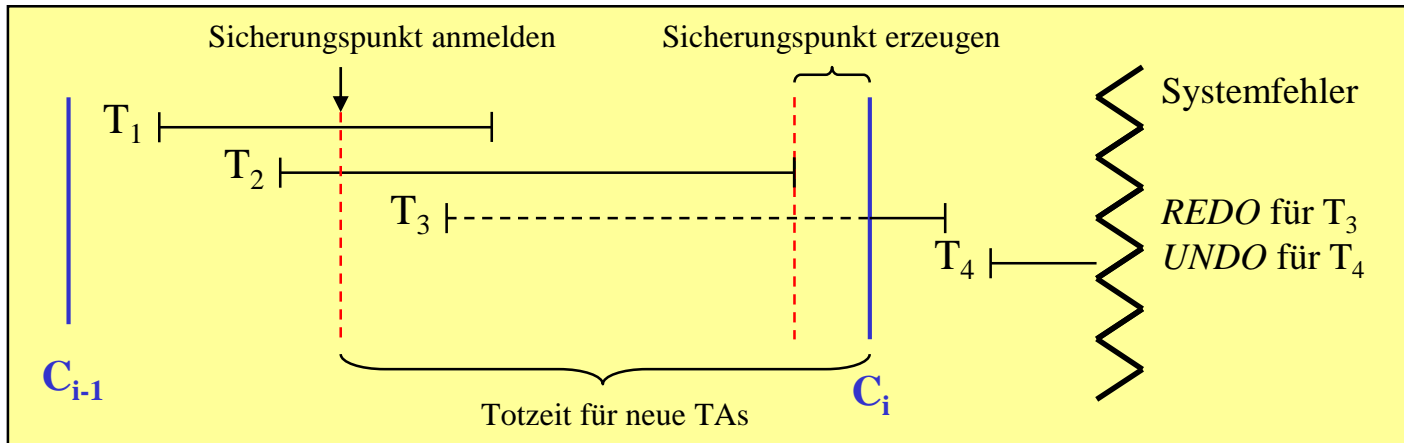
- Vorteile:
 - keine **REDO** nötig
 - Implementierung ist einfach in Kombination mit Seitensperren
- Nachteil: (sehr) aufwändiger Normalbetrieb, insbesondere für Hot-Spot-Seiten
- Beispiel: Sicherungspunkte bei COMMIT von T1 und T2, deshalb kein **REDO** nötig



TCC: TA-konsistente Sicherungspunkte

- DB wird in TA-konsistenten Zustand gebracht, d.h. keine schmutzigen Änderungen
- Während Sicherung keine aktiven Änderungs-TAs
- Sicherungspunkt bezieht sich immer auf alle TAs
- **UNDO-** und **REDO-**Recovery sind durch letzten Sicherungspunkt begrenzt
- Ablauf:
 - Anmeldung des Sicherungspunktes
 - Warten, bis alle Änderungs-TAs abgeschlossen sind
 - Erzeugen des Sicherungspunktes
 - Verzögerung neuer Änderungs-TAs bis zum Abschluss der Sicherung

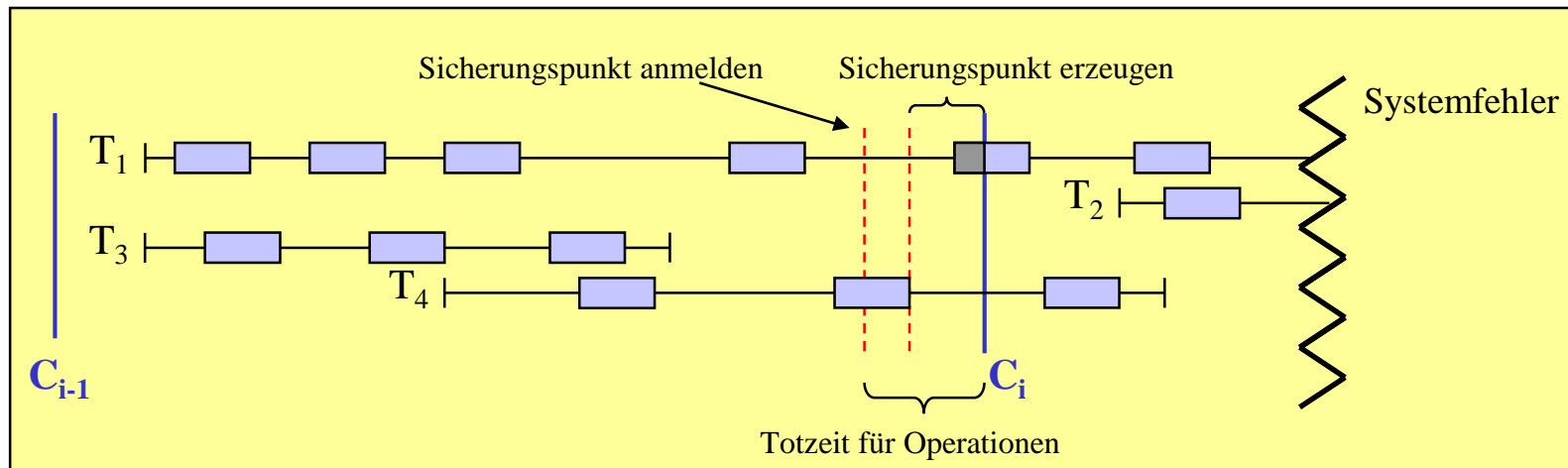
- Vorteil: **UNDO**- und **REDO**-Recovery beginnen beim letzten Sicherungspunkt (im Beispiel: C_i), d.h. es sind nur TAs betroffen, die nach der letzten Sicherung gestartet wurden (hier: T_3, T_4)
- Nachteil: lange Wartezeiten (“Totzeiten”) im System
- Beispiel:



ACC: Aktions-konsistente Sicherungspunkte

- Blockierung nur auf Operationenebene, nicht mehr für ganze TAs
- Keine Änderungsoperationen während der Sicherung
- **UNDO**-Recovery beginnt bei $MinLSN$ = kleinste LSN aller noch aktiven TAs des letzten Sicherungspunktes
- **REDO**-Recovery durch letzten SP begrenzt
- Ablauf:
 - Anmelden des Sicherungspunktes
 - Beendigung aller laufenden Änderungsoperationen abwarten
 - Erzeugen des Sicherungspunktes
 - Verzögerung neuer Änderungsoperationen bis zum Abschluss der Sicherung

- Vorteil: Totzeit des Systems für Änderungen deutlich reduziert
- Nachteil: Geringere Qualität der Sicherungspunkte
 - schmutzige Änderungen können in die Datenbank gelangen
 - zwar **REDO**-, nicht jedoch **UNDO**-Recovery durch letzten Sicherungspunkt begrenzt
- Beispiel:

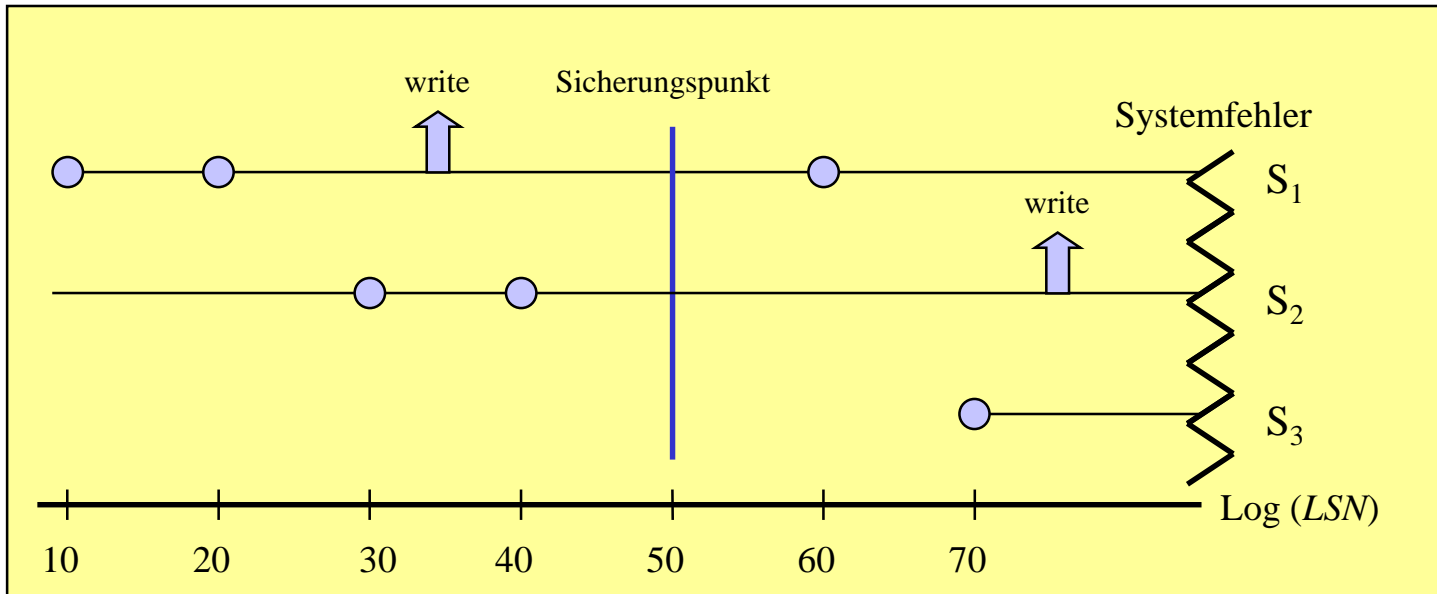


Indirekte Sicherungspunkte

- Charakterisierung
 - Direkte Sicherungspunkte: hoher Aufwand bei großen DB-Puffern nicht akzeptabel
 - Indirekte Sicherungspunkte: Änderungen werden nicht vollständig ausgeschrieben
 - DB hat keinen Aktions- oder TA-konsistenten Zustand, sondern unscharfen (fuzzy) Zustand
- Erzeugung eines indirekten Sicherungspunktes
 - im wesentlichen Logging des Status von laufenden TAs und geänderten Seiten
 - minimaler Schreibaufwand, keine nennenswerte Unterbrechung des Betriebs

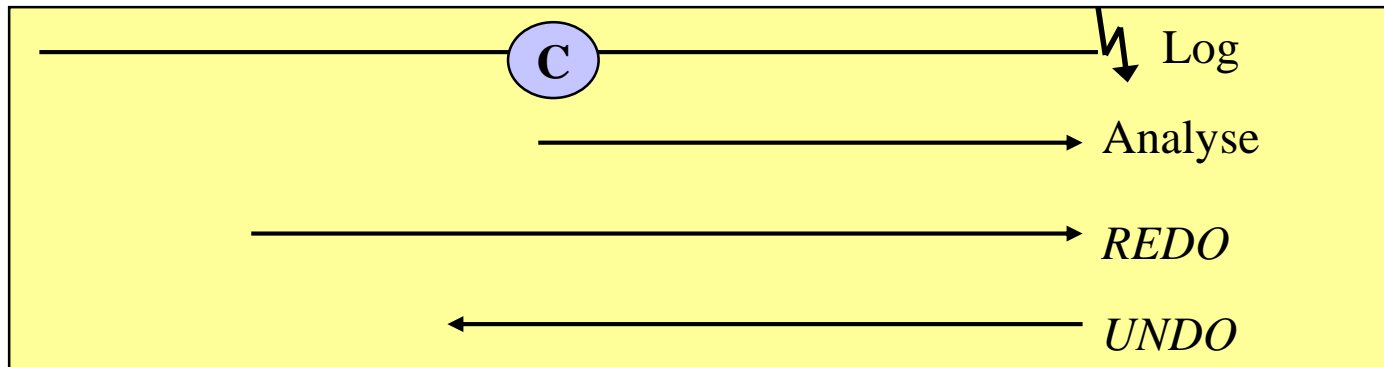
- Ausschreiben von DB-Änderungen
 - außerhalb der Sicherungspunkte, asynchron zur laufenden TA-Verarbeitung
 - länger nicht mehr referenzierte Seiten werden vorausschauend ausgeschrieben
 - Sonderbehandlung für Hot-Spot-Seiten nötig:
 - zwangsweises Ausschreiben bei bestimmtem Log-Umfang
 - Anlegen einer Kopie, um keine Verzögerung für neue Änderungen zu verursachen
- **UNDO**-Recovery beginnt bei *MinLSN* (siehe ACC)
- **REDO**-Recovery
 - Startpunkt ist nicht mehr durch letzten Sicherungspunkt gegeben, auch weiter zurückliegende Änderungen müssen ggf. wiederholt werden
 - Zu jeder geänderten Seite wird *StartLSN* vermerkt (*LSN* der 1. Änderung seit Einlesen von Platte)
 - **REDO** beginnt bei $MinDirtyPageLSN = \min (StartLSN)$

- Beispiel:



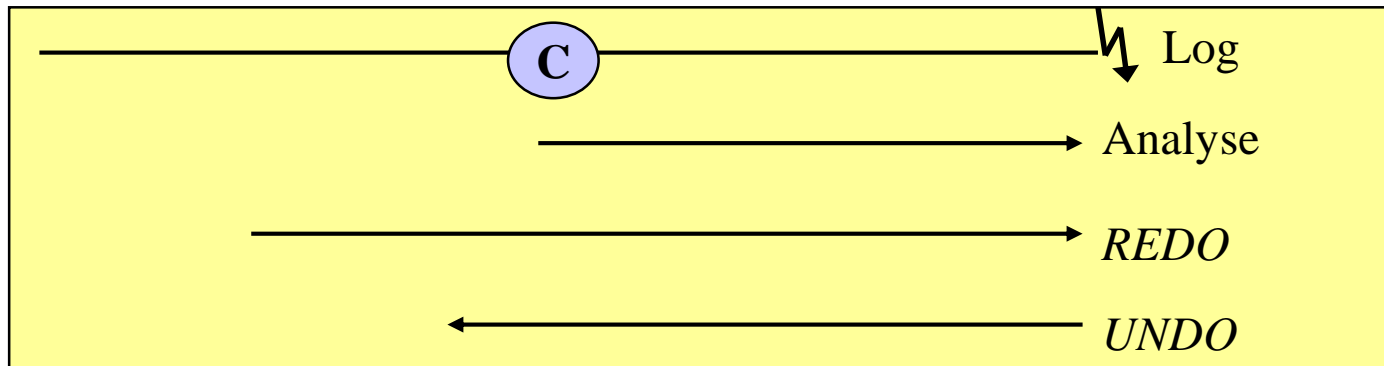
- beim Sicherungspunkt stehen S_1 und S_2 geändert im Puffer
- älteste noch nicht ausgeschriebene Änderung ist auf Seite S_2
- *MinDirtyPageLSN* hat also den Wert 30, dort muss **REDO**-Recovery beginnen

Allgemeine Prozedur der Crash-Recovery



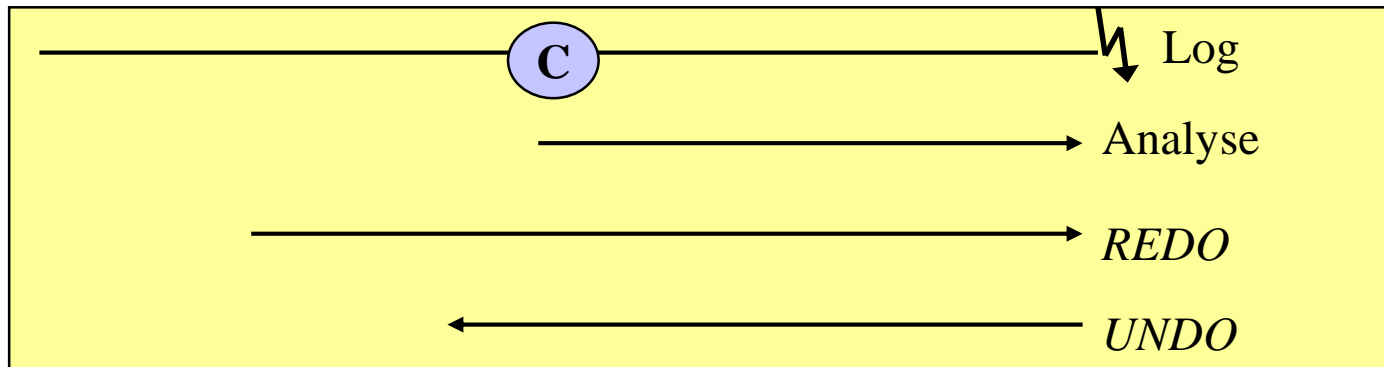
1. Analyse-Phase

- Lies Log-Datei vom letzten Sicherungspunkt bis zum Ende
- Bestimmung von Gewinner- und Verlierer-TAs, sowie der Seiten, die von ihnen geändert wurden
 - Gewinner: TAs, für die ein COMMIT-Satz im Log vorliegt
 - Verlierer: TAs, für die ein ROLLBACK-Satz bzw. kein COMMIT-Satz vorliegt
- Ermittle alle weiteren Seiten, die nach dem Checkpoint geändert wurden



2. REDO-Phase

- Vorwärtslesen der Log-Datei: Startpunkt ist abhängig vom Sicherungspunkttyp
- Aufgabe: Wiederholen der Änderungen, die noch nicht in der DB vorliegen
- zwei Ansätze:
 - vollständiges **REDO** (redo all): Alle Änderungen werden wiederholt
 - selektives **REDO**: Nur die Änderungen der Gewinner-TAs werden wiederholt



3. UNDO-Phase

- Rückwärtslesen der Log-Datei bis BOT der ältesten Verlierer TA
- Aufgabe: Zurücksetzen der Verlierer-TAs
- Fertig wenn Beginn der ältesten TA erreicht ist, die bei letztem Checkpoint aktiv war
- abhängig von REDO-Vorgehen:
 - vollständiges **REDO**: nur zum Fehlerzeitpunkt noch laufende TAs zurücksetzen
 - selektives **REDO**: **alle** Verlierer-TAs zurücksetzen

4. Abschluß der Recovery durch einen Sicherungspunkt

Recovery der Recovery

- Nach einem Fehler während der Recovery beginnt die Recovery der Recovery wieder von vorne (Analysephase, **REDO**, **UNDO**, Sicherungspunkt)
- **REDO**- und **UNDO**-Phasen müssen *idempotent* sein
- Auch bei mehrfacher Ausführung müssen **REDO/UNDO** immer wieder dasselbe Ergebnis liefern
- D.h. zu jeder Änderungsaktion A muss gelten

$$\text{UNDO}(\text{UNDO}(\dots\text{UNDO}(A)\dots)) = \text{UNDO}(A)$$

$$\text{REDO}(\text{REDO}(\dots\text{REDO}(A)\dots)) = \text{REDO}(A)$$

Idempotenz der REDO-Phase

- Für jeden Log-Eintrag E , für den ein **REDO** (tatsächlich) durchgeführt wurde, wird die LSN von E in die (betroffene) Seite eingetragen
- D.h. zu jeder Seite wird protokolliert, welche **REDO**-Operation als letztes ausgeführt wurde
- Verhindert, dass nach einem Absturz während der **REDO**-Phase, das erneute **REDO** nicht versehentlich auf dem AFIM aufsetzt

Idempotenz der UNDO-Phase

- Für jede ausgeführte **UNDO**-Operation wird **ein Compensation Log Record (CLR)** angelegt, der folgende Informationen enthalten muss:
 - Eindeutige Log Sequence Number (*LSN*)
 - ID der beteiligten TA
 - ID(s) der geänderten Seit(en)
 - **REDO**-Information: entspricht der **UNDO**-Operation, die ausgeführt wurde
 - Nach einem Fehler während einer **UNDO**-Operation wird diese Operation dann in der **REDO**-Phase ausgeführt und in der nachfolgenden **UNDO**-Phase übersprungen; dazu enthält jeder CLR einen Pointer zur *LSN* der zu dieser TA gehörenden Änderung, die der kompensierten Operation vorausging (relativ einfach aus *PrevLSN*-Einträgen zu ermitteln)