Kapitel 3 Datensicherheit (Recovery)

- Aufgabe
  - Dauerhafte und konsistente Verfügbarkeit des Datenbestandes sicherstellen.
  - Berücksichtigung möglicher Fehler im laufenden Betrieb und auf den Datenträgern.
  - Wiederherstellung eines konsistenten DB-Zustandes nach einem Fehler (Recovery).
  - ACID-Prinzipien *Atomarität*, *Konsistenz* und *Dauerhaftigkeit.*

- Speichermedien
  - Permanente (materialisierte) Datenbank (wahlfreier Zugriff) => Platten
  - Archivkopien (Backupdateien) => Bänder, optische Platten (auch WORM)

  - temporäre Logdatei (sequentielles Schreiben) => Platte
  - Archivlogdatei => Bänder, optische Platten (auch WORM)

  WORM = "Write Once Read Many"-Speichermedium (z.B. optische Platten)

3.1 Klassifikation von Fehlern

Transaktionsfehler

- Mögliche Ursachen für Abbruch einzelner Transaktionen:
  - explizites Rücksetzen von Transaktionen durch ROLLBACK
  - Fehler der Anwendungsprogramme (z.B. Division durch Null, …)
  - Verletzung von Integritätsbedingungen oder Zugriffsrechten
  - Rücksetzung aufgrund von Synchronisationskonflikten
- Anforderungen zur Behandlung
  - *Undo-Recovery*: Rücknahme der Änderungen abgebrochener TAs.
  - isoliertes Zurücksetzen einzelner TAs soll laufenden Betrieb nicht zu sehr stören.
  - Sperren und andere Ressourcen sollen schnell für andere TAs freigegeben werden.
  - *UNDO* kann sehr häufig auftreten, deswegen schnelle Ausführung wichtig
  - Hilfsmittel: TA-Puffer, temporäre Logdatei
- Probleme mit langen Transaktionen
  - vollständiges Zurücksetzen bringt hohen Arbeitsverlust.
  - Abhilfe durch transaktionsinterne Rücksetzpunkte.
  - partielles Zurücksetzen als Erweiterung (Auflösung) der Atomarität.

Systemfehler

- Charakterisierung
  - Betrieb des DBS ist unterbrochen.
  - Verlust von Hauptspeicherinformation.
  - permanente Speicher (Platten) sind nicht betroffen.

- Mögliche Ursachen
  - Hardwarefehler: Ausfall der CPU
  - Softwarefehler: Absturz des Betriebssystems oder des DBMS
  - Umgebungsfehler: Stromausfall

- Behandlung durch *Crash-Recovery*
  - Ziel: Wiederherstellung des jüngsten transaktionskonsistenten Datenbankzustandes.
  - alle zum Fehlerzeitpunkt laufenden TAs sind betroffen.
  - *Undo-Recovery*: Rücknahme der Änderungen aller nicht erfolgreich beendeten TAs.
  - *Redo-Recovery*: Wiederholung der Änderungen aller erfolgreich beendeten TAs, die aufgrund des Fehlers noch nicht permanent gespeichert waren.
  - Hilfsmittel: Permanente Datenbank, temporäre Logdatei.

Medienfehler

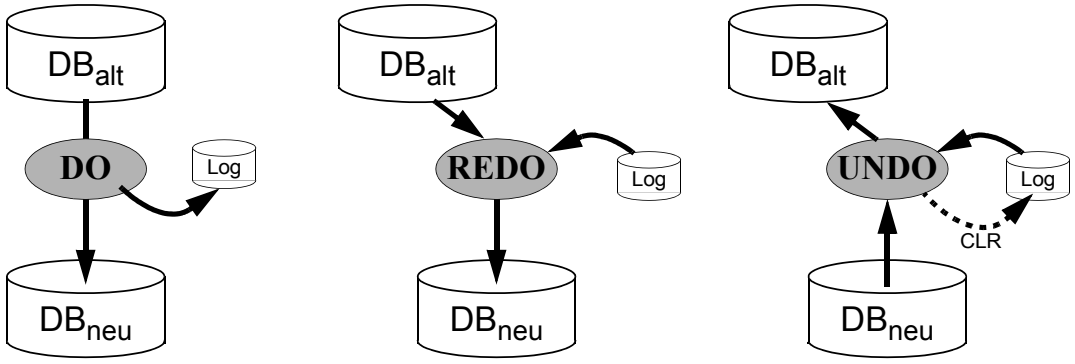
- Charakterisierung
  Verlust von permanenten Daten aufgrund Zerstörung des Speichermediums.
  - Gerätefehler: “Head Crash” bei Magnetplatten.
  - Fehler in Systemprogrammen (z.B. Plattentreiber), die zu Datenverlust führen.
  - Katastrophe: Erdbeben, Brand, Wasserschaden, etc.

- Behandlung durch *Geräte-Recovery*
  - Ziel: Wiederherstellung des jüngstmöglichen transaktionskonsistente DB-Zustands.
  - vor allem *Redo-Recovery* zur Rekonstruktion verlorengegangener Änderungen.
  - Ablauf: Archivkopie einspielen, Einträge im Archivlog nachfahren.
  - Hilfsmittel: Archivkopien, Archivlogdateien.

- Weitergehende Behandlung: *Katastrophen-Recovery*
  - räumlich getrennte Aufbewahrung der Archive wichtig.
  - evtl. Replikation des ganzen Datenbanksystems (z.B. in anderer Stadt).
  - ggf. COMMIT erst dann beenden, wenn Duplikatsystem die Änderung bestätigt.

3.2 Logging-Techniken

- Für jede Änderungsoperation auf der Datenbank benötigt man Protokolleinträge zum:
  — REDO: Information zum Nachvollziehen der Änderungen erfolgreicher TAs.
  — UNDO: Information zum Zurücknehmen der Änderungen unvollständiger TAs.



- CLR: *Compensation Log Records* zur Behandlung von Fehlern während des Recovery.
- Log-Techniken: Physisches Logging, logisches Logging und physiologisches Logging.

Physisches Logging

- Protokoll auf der Ebene der physischen Objekte (Seiten, Datensätze, Indexeinträge); zwei Formen:
  — *Zustandslogging*: Protokollierung der Werte vor und nach der Änderung.
  — *Übergangslogging*: Protokollierung der Zustandsdifferenz.

Zustandslogging

— *Before-Image* (BFIM): ursprünglicher Wert eines Objektes vor der Änderung.
— *After-Image* (AFIM): neuer Wert eines Objektes nach der Änderung.
— bei Mehrfachänderungen nur erstes BFIM und letztes AFIM zu protokollieren.
— zwei Formen: *Seitenlogging* und *Eintragslogging*.

- Zustandslogging auf Seitenebene
  — vollständige Kopien von Seiten werden protokolliert.
  — Recovery sehr einfach und schnell, da Seiten einfach zurückkopiert werden.
  — 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 mindestens eine 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

— Reduziertes Log: speichere statt BFIMs und AFIMs nur "Differenzen"
— Aus BFIM muß AFIM berechenbar sein und umgekehrt.
— Realisierbar durch XOR-Operation $\oplus$ (eXclusive-OR).
— Auf *Seitenebene* Komprimierung möglich wegen vieler '0'.
— Auf *Eintragsebene* nur einfach, falls Länge der Einträge unverändert.

XOR:
$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

Prinzip

	Zustands-Logging	Differenzen-Logging
DO Änderung A_{alt} –> 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 AFIM	$A_{neu} := A_{alt} \oplus D$
UNDO (in DB liegt A_{neu})	Überschreibe A_{neu} mit BFIM	$A_{alt} := A_{neu} \oplus D$

Logisches Logging

- Spezielle Form des Übergangs-Logging
  — nicht physische Zustandsänderungen protokollieren, sondern …
  — Änderungsoperationen mit ihren aktuellen Parametern
- Vorteil
  — Protokoll auf hoher Abstraktionsebene ermöglicht kurze Log-Einträge
- Probleme für REDO
  Änderungen umfassen typischerweise mehrere Seiten (Tabelle, Indexe).
  — Atomares Einbringen der Mehrfachänderungen schwierig.
  — nicht kombinierbar mit Update-in-Place wegen möglicher Schreibunterbrechungen.
  — 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

- *Physical-to-a-page*
  - Protokollierungseinheiten sind geänderte Seiten.
  - gut verträglich mit Pufferverwaltung und direktem Einbringen.
- *Logical-within-a-page*
  - logische Protokollierung der Änderungen auf einer Seite.
  - kein Zusatzaufwand zum Logging von Verwaltungsinformation am Seitenanfang.
  - Löschen und Einfügen ist nicht auf Position innerhalb der Seite festgelegt.

- Bewertung
  - Log-Einträge beziehen sich nicht auf mehrere Seiten wie bei logischem Logging.
  - Einfacheres Recovery als bei logischem Logging (wegen Seitenbezug).
  - Logdatei länger als bei logischem Logging aber kürzer als bei physischem Logging.
  - Flexibler als physisches Logging wegen variabler Objektpositionen auf Seiten.

Struktur von Log-Einträgen

- Art der Protokolleinträge
  - Beginn, Commit und Rollback von Transaktionen.
  - Änderungen des DB-Zustandes durch Transaktionen (REDO/UNDO-Information).
  - Sicherungspunkte (Checkpoints).

- Allgemeine Protokollinformation
  - *Log Sequence Number* (*LSN*): eindeutige Kennung in chronologischer Reihenfolge.
  - *TA-Id*: eindeutige Kennung der Transaktion.
  - *PageId*: Kennung der betroffenen Seite (eigener Eintrag pro geänderter Seite).
  - *PrevLSN*: Rückwärtsverkettung der Einträge ein- und derselben Transaktion.

- Bemerkungen
  - Die *LSN* werden in aufsteigender Reihenfolge vergeben, sodaß sich der zeitliche Ablauf der protokollierten Änderungen nachvollziehen läßt.
  - Die verschiedenen Sätze in der Log-Datei haben unterschiedliche bzw. variable Länge (insbesondere REDO/UNDO-Information bei DB-Änderungen).

Beispiel einer Log-Datei (hier: logisches Logging)

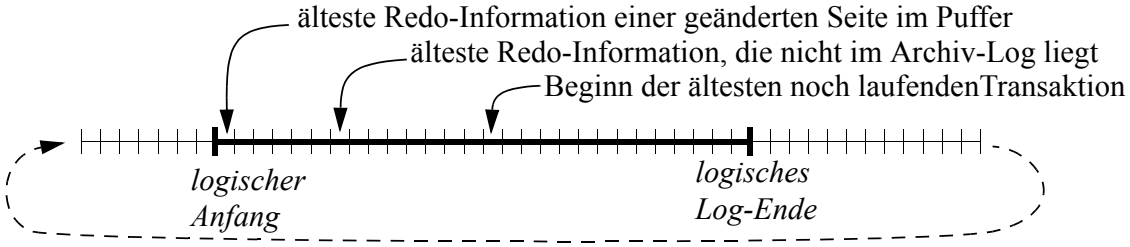
Ablauf T_1	Ablauf T_2	Log-Eintrag (LSN, TA-Id, PageId, Redo, Undo, PrevLSN)
begin		(#1, T_1, **begin**, 0)
read(A, a_1)		
	begin	(#2, T_2, **begin**, 0)
	read(C, c_2) // 80	
$a_1 := a_1 - 50$		
write(A, a_1)		(#3, T_1, p_A, A–=50, A+=50, #1)
	$c_2 := 100$	
	write(C, c_2)	(#4, T_2, p_C, C=100, C=80, #2)
read(B, b_1) // 70		
$b_1 := 50$		
write(B, b_1)		(#5, T_1, p_B, B=50, B=70, #3)
commit		(#6, T_1, **commit**, #5)
	read(A, a_1)	
	$a_2 := a_2 - 100$	
	write(A, a_2)	(#7, T_2, p_A, A–=100, A+=100, #4)
	commit	(#8, T_2, **commit**, #7)

Begrenzung der Log-Größe

- Zwei Arten von Logdateien
  - — *temporäres Log* zur Behandlung von Transaktions- und Systemfehlern.
  - — *Archivlog* für Geräterecovery bei Medienfehler.
- Temporäre Logdatei
  - — UNDO-Information für erfolgreich beendete TAs nicht mehr nötig, da diese nicht mehr zurückgesetzt werden können.
  - — REDO-Information wird nach dem Ausschreiben geänderter Seiten in die permanente DB nicht mehr für Crash-Recovery benötigt.
  - — Ringpufferorganisation der Logdatei durch Beschränkung der Log-Größe möglich.
- Archivlogdatei
  - — REDO-Information wird längerfristig noch für Geräterecovery benötigt.
  - — REDO-Information kann zeitlich verzögert von temporärem Log übernommen werden, dadurch keine Behinderung bei vorübergehend hoher Transaktionslast.
  - — Kein Löschen im Archivlog: WORM-Technik benutzbar (z.B. optische Platten).

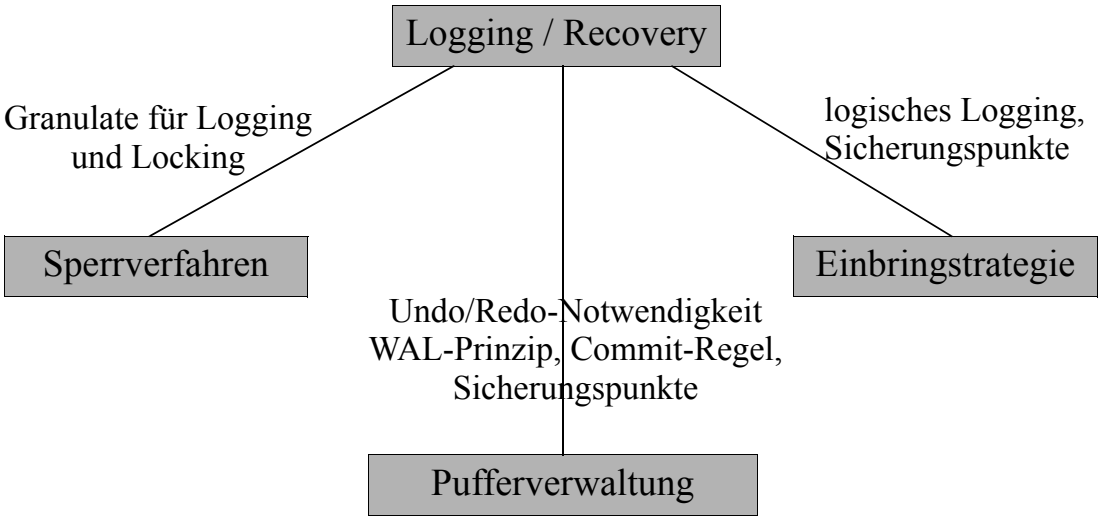
Log-Dateien werden sequentiell erstellt, dadurch schnelles Schreiben möglich.

- Ringpufferorganisation für temporäre Logdatei
  — Die Seiten der Log-Datei werden zyklisch überschrieben



- Inhalt des Ringpuffers
  — Für laufende Transaktionen:
    – *LSN* des ersten Eintrages (**begin**-Eintrag).
    – Minimum (*MinTxLSN*) markiert älteste TA und begrenzt UNDO-Information.
  — Für jede geänderte Seite:
    – *LSN* der ersten Änderung nach dem Einlesen in den Hauptspeicher
    – Minimum (*MinDirtyPageLSN*) begrenzt Redo-Information für Crash-Recovery.
  — LSN der ältesten Redo-Information, die noch nicht auf Archiv-Log liegt.

3.3 Abhängigkeiten von anderen Systemkomponenten



Einbringstrategien

— Direktes Einbringen: Update-in-Place
— Indirektes Einbringen: z.B. Schattenspeichertechnik

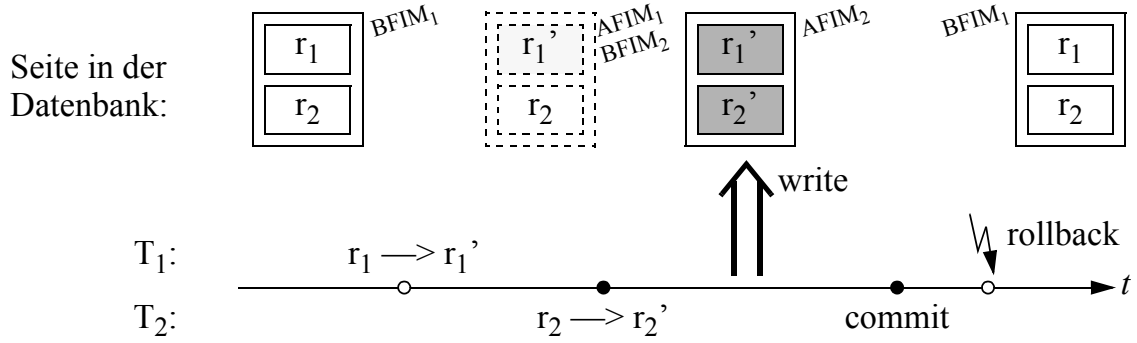
- Direktes Einbringen (*Update-in-Place*)
  — Geänderte Objekte werden auf ihre ursprüngliche Position zurückgeschrieben.
  — Änderungen werden beim Zurückschreiben auf die Platte wirksam.
  — Unterbrechungsfreiheit des Einbringens kann nicht garantiert werden (*NonAtomic*).

- Indirektes (verzögertes) Einbringen
  — Ziel: Einbringen in die Datenbank wird unterbrechungsfrei durchgeführt (*Atomic*).
  — Schreiben auf die Platte und Einbringen in die DB werden unterschieden.
  — Objekte werden auf separaten Seiten in die Datenbank zurückgeschrieben (z.B. *Schattenspeichertechnik* oder *Twin-Block*-Verfahren).
  — Alte Versionen der Objekte bleiben erhalten, d.h. es muß keine UNDO-Information explizit gespeichert werden.
  — Einbringen kann von COMMIT losgelöst werden und z.B. mit Pufferverwaltung gekoppelt werden (z.B. Einbringen bei Sicherungspunkten).
  — Nachteil: Clustereigenschaften der Daten auf der Platte gehen verloren.

Einfluß des Sperrgranulates

- Abhängigkeit zwischen Log-Granulat und Sperrgranulat
  — Log-Granulat muß kleiner/gleich Sperrgranulat sein (sonst Lost Updates möglich).
  — Also: Satzsperren erzwingen feine Log-Granulate (Eintragslogging oder physiologisches Logging).

- Beispiel für Problem bei “Satzsperren mit Seitenlogging”



— T_1 und T_2 ändern die Datensätze r_1 und r_2, die auf derselben DB-Seite liegen.
— Die Seite wird in die DB zurückgeschrieben, T_2 endet mit COMMIT.
— Falls T_1 zurückgesetzt wird, geht auch die Änderung r_2 —> r_2’ verloren
==> Lost Update, eklatanter Verstoß gegen die Dauerhaftigkeit des COMMIT.

Pufferverwaltung: Ausschreiben geänderter Seiten

- Markierung von Seiten
  - *fixed*: Seite wird für Zugriff einer Transaktion im Puffer benötigt.
  - *dirty*: Seite wurde durch eine TA geändert, d.h. $\text{Seite}_{Puffer} \neq \text{Seite}_{DB}$
- Nach dem Zugriff wird die Markierung *fixed* gelöscht.
  - Seite kann zur Verdrängung freigegeben werden.
  - geänderte Seiten (*dirty*) müssen in die DB zurückgeschrieben werden.

Zwei Verdrängungsstrategien: *Steal* und *No-Steal*

- Strategie *No-Steal*
  - Schmutzige Seiten dürfen nicht aus dem Puffer entfernt werden.
  - Persistente Datenbank enthält keine Änderungen nicht-erfolgreicher Transaktionen.
  - UNDO-Recovery ist nicht erforderlich.
- Strategie *Steal*
  - Schmutzige Seiten dürfen ersetzt werden.
  - Datenbank kann unbestätigte Änderungen enthalten.
  - UNDO-Recovery ist erforderlich.

- Nachteile von *No-Steal*
  - Erheblicher Aufwand für die Pufferverwaltung.
  - bei langen Änderungstransaktionen werden große Teile des Puffers blockiert.
  - im Extremfall übersteigt die Anzahl geänderter Seiten die Puffergröße
    ==> No-Steal nicht kombinierbar mit Update-in-Place.
  - DBS verwenden meist flexiblere Variante *Steal* und nehmen UNDO in Kauf.

Zwei Ausschreibestrategien: *Force* und *No-Force*

- Strategie *Force*
  - Geänderte Seiten werden spätestens bei EOT (vor COMMIT) in DB geschrieben.
  - REDO-Recovery ist nicht erforderlich (bei Systemfehler, d.h. Crash-Recovery).
  - sehr hoher I/O-Aufwand, da Änderungen jeder TA einzeln geschrieben werden.
  - Vielzahl an Schreibvorgängen führt zu schlechteren Antwortzeiten, länger gehaltenen Sperren und damit zu mehr Sperrkonflikten.
- Strategie *No-Force*
  - Änderungen können auch nach dem COMMIT erst in die DB geschrieben werden.
  - REDO-Recovery ist erforderlich.
  - Änderungen auf einer Seite über mehrere TAs hinweg können gesammelt werden.
  - Reduzierung des REDO-Aufwandes durch periodische Sicherungspunkte.

Kombination der Verdrängungs- und Ausschreibestrategien

- Vier prinzipielle Möglichkeiten

	No-Steal	*Steal*
Force	kein UNDO – kein REDO (nicht für Update-in-Place)	UNDO – kein REDO
No-Force	kein UNDO – REDO	UNDO – REDO

- 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:
    (1) wegen *No-Steal* dürfen Änderungen erst nach COMMIT in die DB gelangen, was jedoch *Force* widerspricht (*No-Steal => No-Force*)
    (2) 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

Zwei fundamentale Regeln zum korrekten Wiederanlauf im Fehlerfall:

- WAL-Prinzip (*Write-Ahead-Log*)
  - UNDO-Information (z.B. BFIM) muß vor Änderung der DB im Protokoll stehen.
  - wichtig, um schmutzige Änderungen rückgängig machen zu können.
  - nur relevant für STEAL.
  - wichtig bei direktem Einbringen, z.T. aber auch bei indirektem Einbringen nötig.
- COMMIT-Regel (*Force-Log-at-Commit*)
  - REDO-Information (z.B. AFIM) muß vor dem COMMIT im Protokoll stehen.
  - wichtig für REDO bei Crash-Recovery (bei *No-Force*).
  - wichtig für REDO bei Geräte-Recovery (auch bei *Force*).
  - gilt für direkte und indirekte Einbringstrategien gleichermaßen.

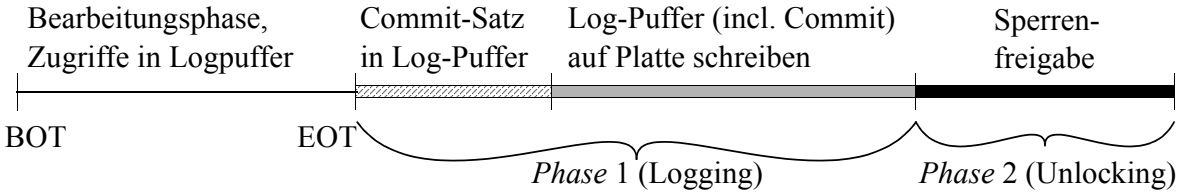
- Bemerkung:
  Um die chronologische Reihenfolge im Ringpuffer zu wahren, werden alle Log-Einträge bis zum letzten notwendigen ausgeschrieben; d.h. es werden keine Log-Einträge übergangen.

COMMIT-Verarbeitung

Generell wird Log-Puffer ausgeschrieben, …

- wenn er vollständig gefüllt ist
- aufgrund der WAL-Regel (UNDO-Info vor DB-Änderungen)
- aufgrund der Commit-Regel (REDO-Info vor Commit)

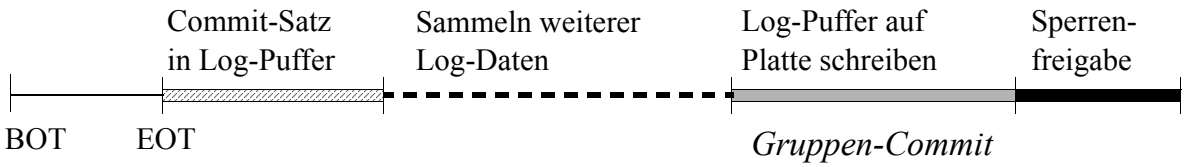
Standard Zwei-Phasen-Commit



- *Phase* 1: *Logging*
  - Überprüfen der verzögerten Integritätsbedingungen
  - Logging der REDO-Informationen incl. Commit-Satz
- *Phase* 2: *Unlocking*
  - Freigabe der Sperren (Sichtbarmachen der Änderungen)
  - Bestätigung des Commit an das Anwendungsprogramm

Gruppen-Commit

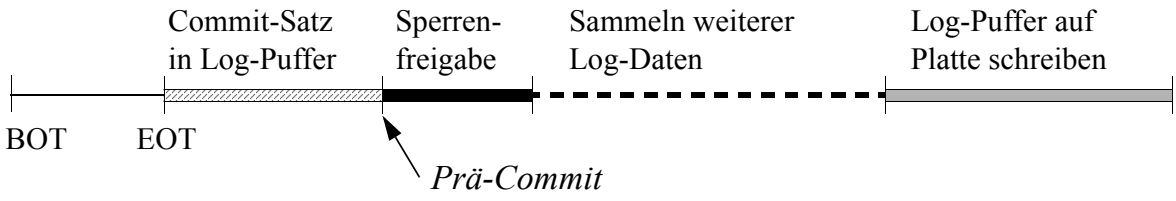
- *Problem*: Commit-Regel verlangt Ausschreiben des Log-Puffers bei jedem Commit
  - Beeinträchtigung für kurze TAs, deren Log-Daten weniger als eine Seite umfassen
  - Durchsatz an TAs ist eingeschränkt
- *Abhilfe*: Gruppen-Commit
  - Log-Daten mehrerer Transaktionen werden im Puffer gesammelt
  - Log wird auf Platte geschrieben, sobald Puffer gefüllt ist oder nach Timeout



- *Bewertung*:
  - Vorteil: Reduktion der Plattenzugriffe und höhere Transaktionsraten möglich
  - Nachteil: längere Sperrdauer führt zu längeren Antwortzeiten
  - wird von zahlreichen DBS unterstützt

Prä-Commit

- Konzept
  - Ziel: lange Sperrzeiten bei Gruppen-Commit sollen vermieden werden
  - Idee: Freigabe der Sperren, sobald Commit-Satz im Logpuffer



- 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

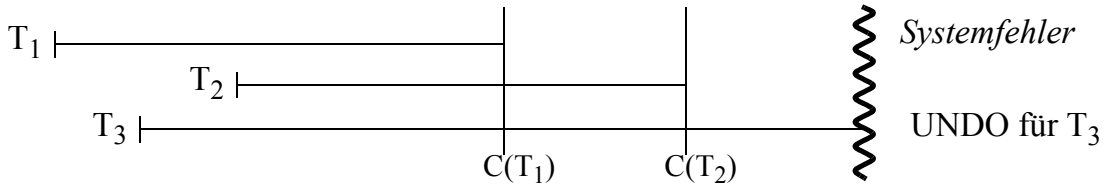
3.4 Sicherungspunkte (Checkpoints)

- Motivation
  - Begrenzung des REDO-Aufwands und des Log-Umfangs
  - Beispiel Hot-Spot-Seiten: werden (fast) nie aus dem Puffer verdrängt
- Direkte Sicherungspunkte
  - alle Änderungen werden in die persistente Datenbank eingebracht
  - alle geänderten Seiten im Hauptspeicher werden auf Platte geschrieben
  - wichtig für indirekte Einbringstrategien (Atomic): Umschalten der Datenbank
- Indirekte (unscharfe, fuzzy) Sicherungspunkte
  - hoher Aufwand des vollständigen Ausschreibens wird vermieden
  - im Log werden im wesentlichen Statusinformationen protokolliert
  - nur für Update-in-Place (NonAtomic) anwendbar; für "Atomic" nicht brauchbar
- Durchführung von Sicherungspunkten
  - Spezielle Log-Einträge: BEGIN_CHKPT, Info über laufende TAs, END_CHKPT
  - LSN des letzten vollständig ausgeführten Checkpoints wird in Restart-Datei geführt
- Häufigkeit von Sicherungspunkten
  - *zu selten*: hohe REDO-Aufwände vs. *zu oft*: hoher Overhead im Normalbetrieb
  - z.B. Sicherungspunkte nach bestimmter Anzahl von Log-Sätzen einfügen

Direkte Sicherungspunkte

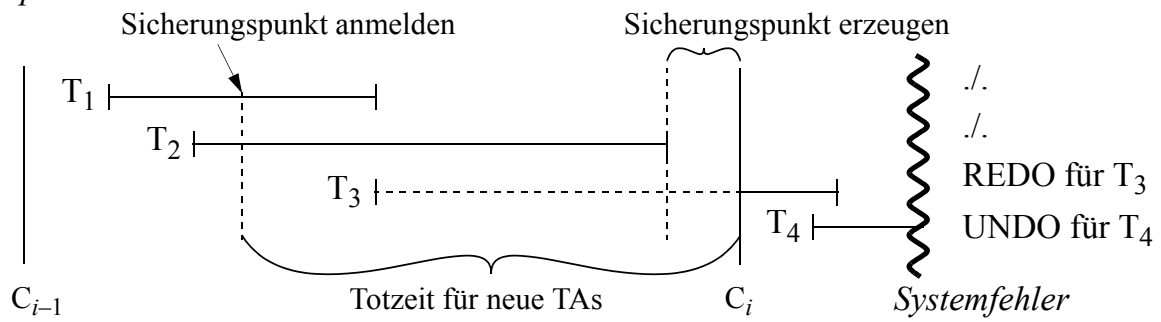
- Charakterisierung
  - alle geänderten Seiten werden in die persistente Datenbank (Platte) geschrieben
  - Zeitbedarf steigt mit dem zeitlichen Abstand der Sicherungspunkte
  - Multi-Page-Access hilft, Schreibkopf-Positionierungen zu minimieren
  - REDO-Recovery kann beim letzten vollständig ausgeführten Checkpoint beginnen
- **Transaktions-orientierte Sicherungspunkte (TOC)**
  - Force-Ausschreibestrategie, d.h. Ausschreiben aller Änderungen beim Commit
  - nicht alle Seiten im Puffer, sondern nur die Änderungen der jeweiligen TA schreiben
  - Implementierung ist einfach in Kombination mit Seitensperren
  - bei Update-in-Place ist UNDO-Recovery nötig, da Force Steal impliziert
  - *Vorteil*: keine REDO-Recovery nötig
  - *Nachteil*: (sehr) aufwändiger Normalbetrieb, insbesondere für "Hot-Spot"-Seiten

  Beispiel: Sicherungspunkte bei Commit von T_1 und T_2, deshalb kein REDO nötig.

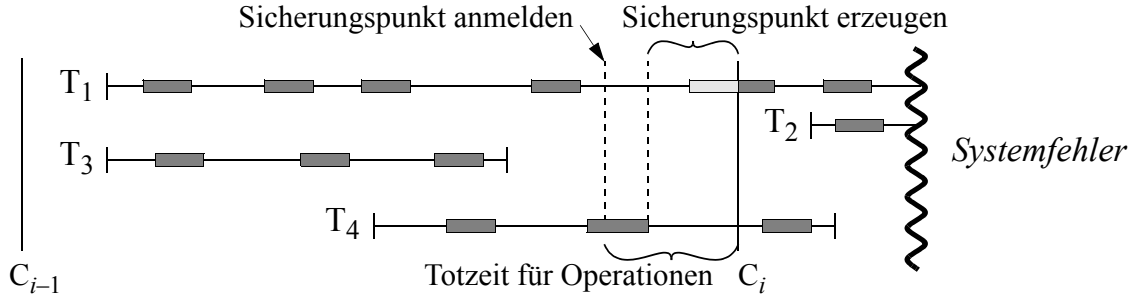


- **Transaktions-konsistente Sicherungspunkte (TCC)**
  - DB wird in TA-konsistenten Zustand gebracht, d.h. keine schmutzigen Änderungen.
  - während der Sicherung dürfen keine Änderungstransaktionen aktiv sein.
  - *Ablauf*:
    - Anmeldung des Sicherungspunktes
    - Warten, bis alle Änderungstransaktionen abgeschlossen sind
    - Erzeugen des Sicherungspunktes
    - Verzögerung neuer Änderungstransaktionen bis zum Abschluß der Sicherung
  - *Vorteil*: Recovery ist durch letzten Sicherungspunkt begrenzt (im Beispiel: C_i), d.h. nur für TAs, die nach der letzten Sicherung gestartet wurden (hier: T_3, T_4)
  - *Nachteil*: lange Wartezeiten ("Totzeiten") im System

  Beispiel:



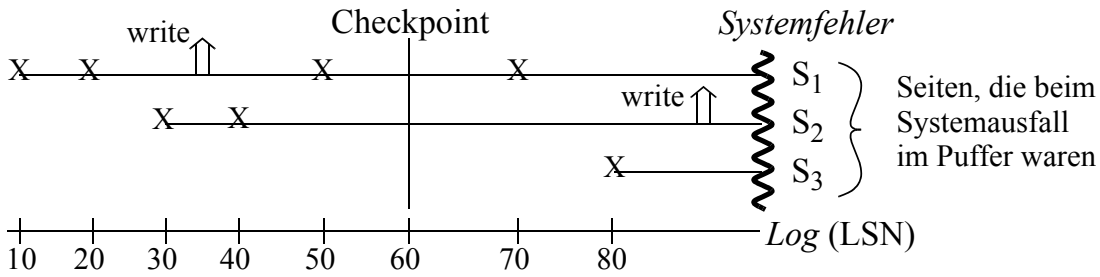
- **Aktions-konsistente Sicherungspunkte (ACC)**
  - Blockierung nur auf Operationenebene, nicht mehr für ganze TAs
  - Keine Änderungsoperationen während der Sicherung
  - *Ablauf*:
    - Anmelden des Sicherungspunktes
    - Beendigung aller laufenden Änderungsoperationen abwarten (im Beispiel: T_4)
    - Erzeugen des Sicherungspunktes
    - Verzögerung neuer Änderungsoperationen bis zum Abschluß der Sicherung (T_1)
  - *Vorteil*: Totzeit des Systems für Änderungen deutlich reduziert
  - *Nachteil*: Geringere Qualität der Sicherungspunkte
    - schmutzige Änderungen können in die Datenbank gelangen (d.h. ACC => Steal)
    - zwar REDO-, nicht jedoch UNDO-Recovery durch Sicherungspunkt begrenzt
  - *Beispiel*:



Fuzzy Checkpoints

- 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* 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, die praktisch nie ersetzt werden:
    - zwangsweises Ausschreiben bei bestimmtem Log-Umfang
    - Anlegen einer Kopie, um keine Verzögerung für neue Änderungen zu verursachen
- REDO-Recovery
  - Startpunkt ist nicht mehr durch letzten Checkpoint gegeben
  - auch weiter zurückliegende Änderungen müssen ggf. wiederholt werden
  - Vorschlag: Seiten ausschreiben, die bei der letzten Sicherung schon geändert waren

- Ergänzend zu REDO: *MinDirtyPageLSN*
  - — Verwalte LSN der ersten Änderung für jede Seite seit letztem Abgleich mit der DB
  - — Minimum dieser LSN bestimmt Logposition, an der REDO-Recovery beginnt
  - — *MinDirtyPageLSN* wird als Teil des Sicherungspunktes in Log-Datei eingetragen
- Beispiel:



  - — beim Checkpoint 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

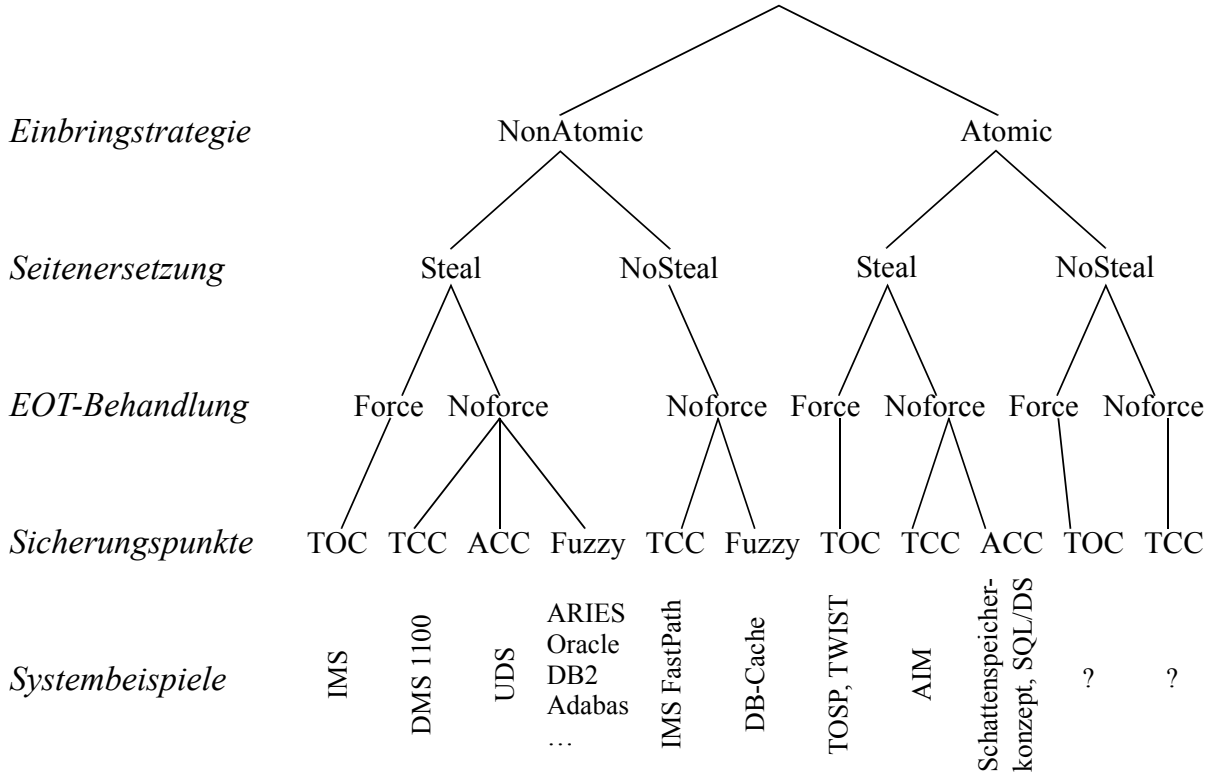
3.5 Recovery-Techniken

- Wiederherstellung im Fehlerfall
  - — *Crash-Recovery* bei Systemfehler:
    Hauptspeicher ist verloren, permanente DB ist erhalten geblieben
  - — *Geräte-Recovery* bei Medienfehler:
    permanente DB (auf Platte) ist beschädigt

- Wichtige Forderung: *Idempotenz*
  - — Recovery muß robust sein gegen Fehler während der Recovery
    d.h. UNDO(…UNDO(p)…) = UNDO(p)
    und REDO(…REDO(p)…) = REDO(p)

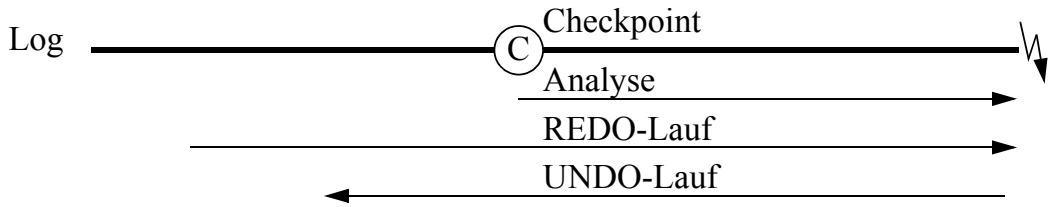
- Abhängigkeiten der Strategien
  Nicht alle Strategien zur Pufferverwaltung etc. lassen sich kombinieren:
  - — NonAtomic, Force => Steal
  - — NonAtomic, No-Steal => No-Force
  - — Force <=> TOC
  - — Atomic => keine Fuzzy Checkpoints
  - — ACC => Steal

Klassifikation von Recovery-Verfahren



Crash-Recovery

- Vorzunehmende Aktionen von der gewählten Recovery-Strategie abhängig
  — Im folgenden: *NonAtomic*, *Steal*, *No-Force*, *Fuzzy Checkpoints*
- Ablauf der Wiederherstellung:
  — Analyse des Logfiles
  — REDO-Phase
  — UNDO-Phase



- **Phase 1**: Analyse des Logfiles
  — Lies Logfile vom letzten Checkpoint bis zum Ende
  — Ermittle TAs, die beim letzten Checkpoint gelaufen sind, sowie geänderte Seiten
  — Davon ausgehend, ermittle Gewinner- und Verlierer-TAs bis zum Systemfehler:
  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

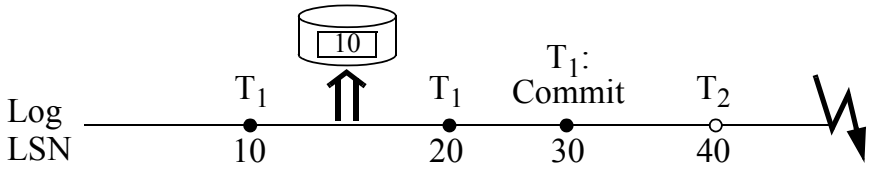
- **Phase 2**: REDO-Lauf
  - Aufgabe: Wiederholen der Änderungen, die noch nicht in der DB vorliegen
  - Startpunkt (bei Fuzzy Checkpoints): *MinDirtyPageLSN* des letzten Checkpoints
  - Vorwärtslesen des Logfiles, um Änderungen zu wiederholen
  - zwei Ansätze:
    – vollständiges REDO (*redo all*): Alle Änderungen werden wiederholt
    – selektives REDO: Nur die Änderungen der Gewinner-TAs werden wiederholt
  - Betroffene Seiten müssen ggf. in den Hauptspeicher geladen werden
- **Phase 3**: UNDO-Lauf
  - Aufgabe: Zurücksetzen der Verlierer-TAs
  - Logfile vom Ende her lesen, um Änderungen umgekehrt zurückzunehmen
  - betroffene Seiten müssen ggf. in den Hauptspeicher geladen werden
  - 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 (beendete und unbeendete)
- Abschluß der Recovery
  - nach der UNDO-Phase wird die Recovery mit einem Checkpoint abgeschlossen

Durchführung des REDO

- Technik
  - Einfach bei physischem und physiologischem Logging wegen Seitenbezogenheit
  - *PageLSN* im Seitenkopf kennzeichnet letzte Speicherung der Seite (“Versionsnr.”)
  - REDO nur erforderlich für Änderungen, deren *LSN* größer als die *PageLSN* ist
  - Anwendung des Logeintrags L auf die Seite B:

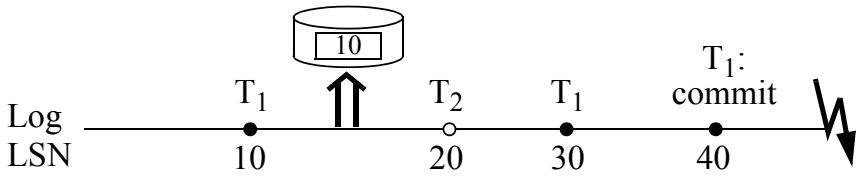
    if (B nicht im Puffer) **then** (lies B in Hauptspeicher ein) **endif**;
    if *LSN*(L) > *PageLSN*(B) **then** REDO (Änderungen aus L); *PageLSN*(B) := *LSN*(L); **endif**;

  - Idempotenz des REDO ist sichergestellt, da bereits wiederholte Änderungen erkannt werden; weitere Wiederholungen werden automatisch vermieden
- Beispiel



  - Änderung der Seite mit LSN=10 wurde vor Commit auf Platte geschrieben
  - nachfolgende Änderungen (#20, #40) werden nur in Puffer und ins Log geschrieben
  - bei Recovery: #10 benötigt kein REDO, #20 schon. *PageLSN* wird auf 20 gesetzt

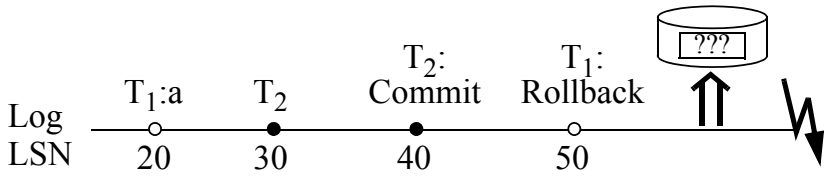
- **Selektives REDO**
  - nur die Änderungen der Gewinner-TAs werden wiederholt (im Beispiel: T_1)
  - UNDO nur für ausgeschriebene Änderungen, d.h. *LSN* ≤ *PageLSN* (also nicht T_2)
  - funktioniert nur bei Seitensperren korrekt
- Problem bei Satzsperren
  - Lost Updates möglich, falls mehrere Sätze auf einer Seite
  - Abhilfe: Verwendung von *LSN* für Sätze statt für Seiten
    => verursacht zu hohen Speicher- und Wartungsaufwand
  - Beispiel: nach REDO von #30 ist *PageLSN*=30; die Änderung #20 wird dann fälschlicherweise als geschrieben betrachtet (#20 ≤ #30) und rückgängig gemacht.



- **Vollständiges REDO**
  - Alle Änderungen von Gewinner- und von Verlierer-TAs werden wiederholt (T_1, T_2)
  - UNDO für alle Änderungen nicht beendeter Verlierer-TAs (im Beispiel: T_2)
  - *PageLSN* wird also nur für REDO, nicht für UNDO herangezogen
  - im Beispiel: Änderungen #20 und #30 werden wiederholt, danach #20 zurückgesetzt

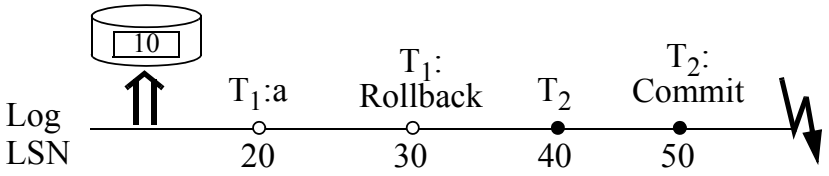
Compensation Log Records (CLR)

- Unterscheide bei Verlierer-Transaktionen:
  - unbeendete TAs (d.h. weder Rollback noch Commit im Log):
    – UNDO wie beschrieben durchführen
  - gescheiterte TAs (d.h. durch Rollback beendet):
    – UNDO ist bereits im Normalbetrieb erfolgt
    – bei Crash-Recovery korrekt berücksichtigen (Idempotenz der Recovery!)
    – welche *PageLSN* soll den betroffenen Seiten zugewiesen werden?
- Probleme bei Satzsperren
  - *PageLSN* darf nicht auf ursprünglichen Wert zurückgesetzt werden
  - Beispiel:



  - würde *PageLSN* auf #20 zurückgesetzt, so ist die Änderung #30 nicht repräsentiert
  - Änderungen bei Rollback sind zu protokollieren; deren LSN an *PageLSN* zuweisen!

- Probleme bei selektivem REDO und Seitensperren
  - alleinige Wiederholung der Gewinner-TAs kann zu Inkonsistenzen führen
  - Beispiel:

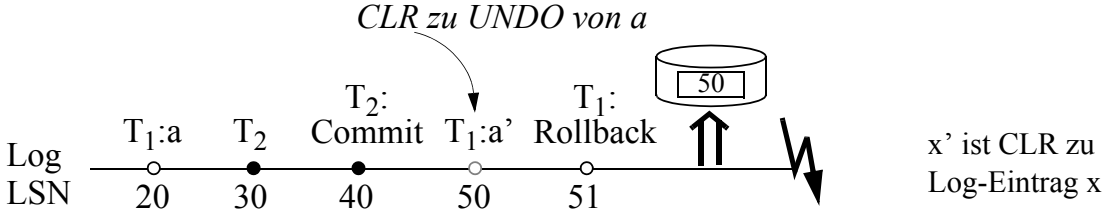


  - in der REDO-Phase wird *PageLSN* auf #40 erhöht
  - in der UNDO-Phase wird die Änderung #20 wegen *LSN* ≤ *PageLSN* als durchgeführt betrachtet und zurückgesetzt, obwohl sie tatsächlich nicht vorliegt
  - UNDO-Operationen durch Rollback müssen also explizit protokolliert werden

- Compensation Log Records (CLR)
  - Protokollierung von UNDO-Operationen während des Rollback im Normalbetrieb
  - für jede zurückgesetzte Änderung wird ein eigener CLR ins Log geschrieben
  - die *LSN* des CLR wird als *PageLSN* für die Seite übernommen
  - Rollback-Satz im Log bestätigt die Rücksetzungen und deren Protokollierung
  - UNDO-Operationen während Recovery sind ebenfalls durch CLR zu protokollieren

CLR bei vollständigem REDO

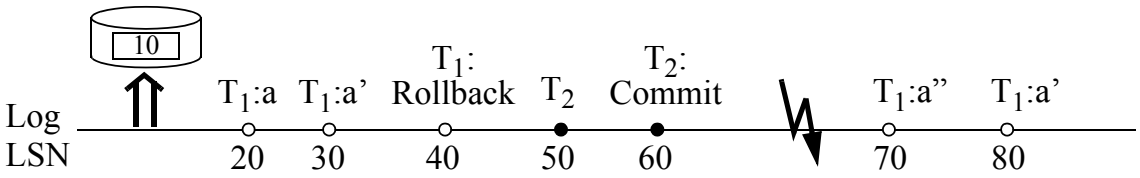
- Vorgehen
  - Wiederholung aller Änderungen von Gewinner- und von Verlierer-TAs
  - für zurückgesetzte TAs werden auch die CLR-Änderungen wiederholt (d.h. REDO früherer UNDOs); danach keine Spuren mehr in der Datenbank
  - in der UNDO-Phase müssen nur noch die unbeendeten TAs zurückgesetzt werden, d.h. für die weder Commit noch Rollback im Log vorliegt.
- obiges Beispiel für “Satzsperren”:
  - bei Rollback von T_1 wurde CLR #50 für UNDO von #20 eingefügt
  - *PageLSN* wurde dabei auf #50 gesetzt
  - Crash-Recovery:
    - REDO-Phase: es müssen keine Änderungen wiederholt werden
    - UNDO-Phase: entfällt, da keine unbeendeten TAs vorliegen



- obiges Beispiel für “Seitensperren”:
  — REDO-Phase: Wiederholung aller Änderungen, incl. #20 und zugehörigem CLR
  — UNDO-Phase: entfällt, da es keine offenen TAs gibt

CLR bei selektivem REDO

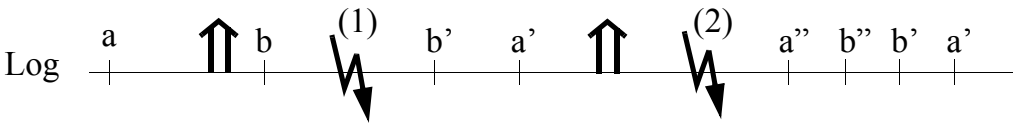
- Vorgehen
  — kein REDO für Änderungen von Verlierer-TAs
  — UNDO für alle Verlierer-TAs, d.h. unbeendete und mit Rollback beendete
- obiges Beispiel mit “Seitensperren”:



  — für UNDO von T_1:a (#20) wird CLR T_1:a’ (#30) protokolliert
  — bei Crash-Recovery wird T_2 (#50) wiederholt, *PageLSN* wird auf #50 gesetzt
  — das UNDO von T_1:a’ und T_1:a wird mit den CLRs #70 und #80 protokolliert

Idempotenz der Crash-Recovery

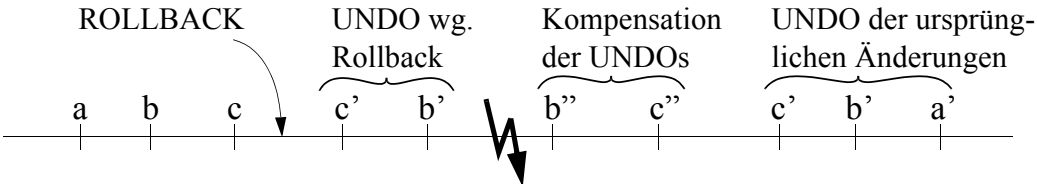
- Notwendigkeit
  — Auch während der Crash-Recovery können Systemfehler auftreten
  — Korrekte Wiederherstellung muß auch im Wiederholungsfall gewährleistet sein
  — Operationen der REDO-Recovery müssen nicht eigens protokolliert werden
  — Operationen der UNDO-Recovery sind durch CLR zu protokollieren
- Beispiel
  — bei selektivem REDO wird UNDO nur vorgenommen, falls $LSN \leq PageLSN$



  — Recovery bei (1):
    – schmutzige Änderung *a* liegt in der DB und wird zurückgenommen
    – schmutzige Änderung *b* liegt nicht in der DB und ist nicht zurückzunehmen
    – CLR *b*’ muß dennoch protokolliert werden, um Fehler (2) korrekt zu behandeln
  — Recovery bei (2):
    – ohne *b*’ würden *a*’, *b* und *a* zurückgesetzt, obwohl *b* nicht auf der Seite liegt
    – mit *b*’ werden also *a*’ (CLR *a*”), *b*’, *b* und *a* korrekt zurückgesetzt
  — bei vollständigem REDO wäre CLR *b*’ ohnehin erforderlich

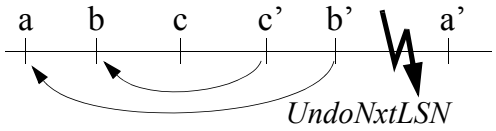
Optimierung des CLR-Einsatzes

- Rücknahme von UNDO-Operationen bei *selektivem* REDO
  - UNDO aufgrund Rollback im Normalbetrieb wird bei Recovery zurückgenommen
  - Systemfehler während Recovery führen zu wiederholten Rücknahmen von UNDOs
- Rücknahme von UNDO-Operationen bei *vollständigem* REDO
  - UNDO aufgrund Rollback wird gemäß CLR in der REDO-Phase wiederholt und in der UNDO-Phase dann nicht mehr berücksichtigt
  - durch Systemfehler abgebrochene Rücksetzungen müssen kompensiert werden
- Beispiel (hier: selektives REDO):



- Optimierung des Log
  - Rücknahme von Rücknahmen ist offensichtlich wenig elegant
  - wiederholte Kompensation stellt unnötigen Aufwand dar
  - Optimierung möglich: UNDO ist erledigt, wenn REDO des CLR bearbeitet ist
  - UNDO also nur für Änderungen erforderlich, zu denen kein CLR vorliegt

- Vorgehen
  - Verweis *UndoNxtLSN* zeigt auf Vorgänger der zurückgesetzten Änderung
  - bei Recovery wird ein CLR nicht kompensiert, sondern mit der Rücksetzung der Änderung fortgefahren, auf die *UndoNxtLSN* zeigt
  - gibt es zu CLR *x'* keinen Vorgänger *UndoNxtLSN*:
    - *x* war die erste Operation der Transaktion
    - Rollback ist erfolgreich abgeschlossen
  - die Optimierung garantiert, daß jede Änderung höchstens einmal zurückgesetzt wird
- obiges Beispiel nach Optimierung des Log:
  - nach Systemfehler wird direkt mit UNDO von *a* fortgefahren:

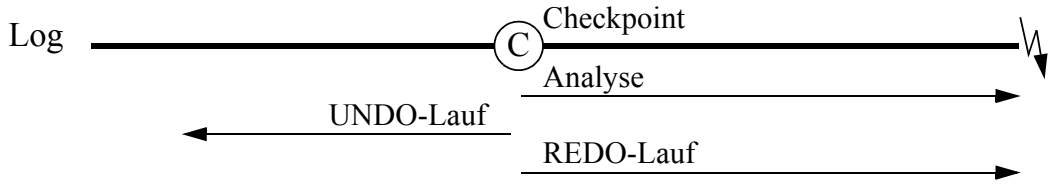


- CLR im Log enthält folgende Informationen:
  - *LSN*, TA-Id, Page-Id, REDO-Info, *PrevLSN*, *UndoNxtLSN*
  - REDO-Information im CLR entspricht UNDO der ursprünglichen Änderung
  - UNDO-Information wird nicht benötigt, stattdessen *UndoNxtLSN*

- Vorteile des vollständigen REDO gegenüber selektivem REDO
  - bessere Nutzbarkeit der CLRs
  - Unterstützung von Satzsperren

Beispiel: Crash-Recovery bei direkten Sicherungspunkten

- Hier: *Atomic*, *Steal*, *Noforce*, *ACC*, logisches Logging, Seitensperren

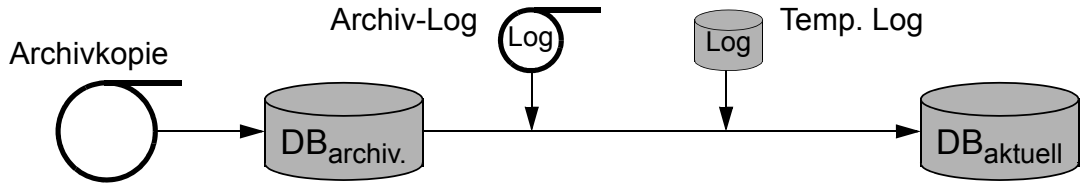


- UNDO-Lauf ab Sicherungspunkt rückwärts bis zum Beginn des ältesteten Verlierers
- REDO-Lauf ab Sicherungspunkt selektiv möglich
- LSNs werden nicht beachtet, Idempotenz durch Starten bei Checkpoint garantiert
- hoher Aufwand für REDO- und UNDO-Operationen

3.6 Geräte-Recovery

- Charakterisierung
  - Behandlung von Medienfehlern auf Sekundärspeichern (z.B. Plattencrash)
  - Persistente, materialisierte Datenbank geht verloren
  - Verwendung von Archivkopien und Archiv-Logdateien
- Arbeiten mit Archivdateien
  - Archivkopie (*Backup*, *Dump*) ist Schnappschuß der Datenbank (z.B. periodisch)
  - Archivlog protokolliert DB-Änderungen seit letztem Backup
  - Log-Sätze werden aus temporärem Log unverändert übernommen
  - Kompaktifizierung des Log: Nur REDO-Informationen erfolgreicher TAs kopieren (d.h. keine UNDO-Informationen, sowie kein REDO erfolgloser TAs)
- Granularität der Archivierung
  - statt gesamter DB evtl. einzelne DB-Segmente archivieren
  - Archivierungsfrequenz läßt sich an Änderungshäufigkeit der Segmente anpassen
  - Vielzahl an Archivkopien und Logdateien erhöht Verwaltungskomplexität

- Ablauf der Geräte-Recovery



  - Einspielen der letzten verfügbaren Archivkopie
  - REDO der Änderungen aus dem Archivlog
  - Ergänzendes REDO erfolgreicher TAs vom temporären Log

- REDO bei Geräte-Recovery
  - grundsätzlich wie bei Crash-Recovery
  - bei Update-in-Place wird über *PageLSN* entschieden, ob Log-Satz anzuwenden ist
  - bei Kompaktifizierung des Archivlogs kein Analyselauf über dem Log nötig
  - bei TA-konsistenten Backups genügt selektives REDO (falls Log nicht kompaktif.)
  - bei “Fuzzy Dumping” auch vollständiges REDO mit UNDO/CLR möglich

- Strategien zur Archivierung
  - *vollständige Archivierung*: alle Seiten der DB (bzw. des Segments) werden kopiert
  - *inkrementelle Archivierung*: nur diejenigen Seiten werden archiviert, die seit der letzten Archivierung geändert wurden

Vollständige Archivierung

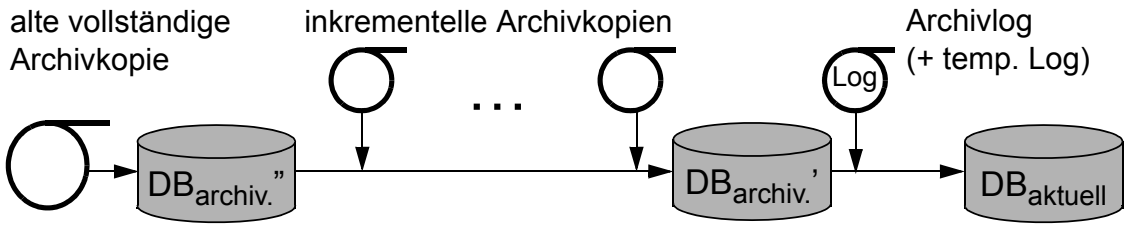
- Eigenschaften
  - vollständiges Kopieren der gesamten DB ist sehr aufwändig
  - Ausnutzen von sequenziellem I/O, Prefetching und Parallelisierung möglich
  - Archivierungszeiten liegen oft im Stundenbereich

- Fuzzy-Dumping zur Online-Archivierung
  - keine Konsistenzanforderung, beliebige Änderungszustände werden archiviert
  - Aktuelle LSN wird zu Beginn der Archivierung vermerkt
  - Archivierung kann als spezielle Lesetransaktion aufgefaßt werden
  - ohne Lesesperren: schmutzige Änderungen im Archiv, UNDO nötig (z.B. durch Anwendung der CLR bei vollständigem REDO)
  - bei kurzen Lesesperren keine schmutzige Änderungen, selektives REDO möglich
  - Synchronisationskonflikte der Archivierung mit Änderungs-TAs möglich

- Transaktions-konsistente Archivierung
  - erreichbar durch lange Lesesperren auf der DB (bzw. dem DB-Segment)
  - Änderungsbetrieb kommt dabei aber vollständig zum Erliegen
  - Vorschlag “Copy on Update”: Before-Images statt geänderter Seiten ins Archiv
    - Archiv spiegelt Zustand vor der Änderung wieder
    - geringer Zusatzaufwand zur Erstellung der BFIMs

Inkrementelle Archivierung

- Vorgehen
  - nur die seit der letzten Archivierung geänderten Seiten werden archiviert
  - wesentlich geringerer Schreibaufwand als bei vollständiger Archivierung
  - dafür ist Geräte-Recovery aufwändiger, da mehrere Archivkopien gelesen werden
- Ablauf der Recovery



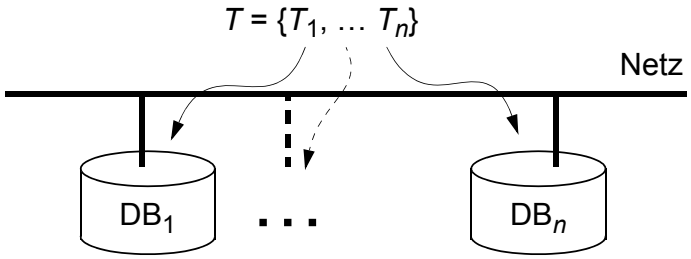
  - letzte vollständige Archivkopie wird eingespielt
  - die Seiten aus nachfolgend erstellten inkrementellen Archivkopien werden ersetzt
  - zuletzt werden die Änderungen aus dem Log nachgezogen (REDO)
  - Reduktion des Recovery-Aufwands möglich: inkrementelle Archivkopien können neben dem laufenden Betrieb in vollständige Archivkopien eingearbeitet werden

Alternativen zur Geräte-Recovery

- Geräte-Recovery ist sehr zeitaufwändig
  - nach der Erkennung des Gerätefehlers wird zuerst das Medium ausgetauscht
  - Einspielen der Archivkopien meistens manuell
  - Verfügbarkeit des Systems dabei stark eingeschränkt; andere Techniken erwünscht
- Spiegelplatten
  - alle Daten werden auf unabhängigen Platten doppelt geführt und aktualisiert
  - bei einfachem Plattenfehler keine Unterbrechung des Betriebs nötig
  - Nachteil: verdoppelte Speicherkosten
  - doppeltes Schreiben kein Problem, da parallel und asynchron (bei Noforce) möglich
  - Lesezugriffe schneller als bei einfachen Platten, da weniger Armbewegungen nötig
- Disk-Arrays (RAID)
  - bessere Speichernutzung durch ausgeklügeltere Redundanz (Faktor < 2)
  - hohe Ausfallsicherheit (Datenverlust nur beim Ausfall zweier Platten)
  - Online-Fehlerbehandlung dafür sehr aufwändig
- Geographisch entferntes Duplikatsystem
  - hilft in Katastrophenfällen (Flugzeugabsturz, Erdbeben) bei totalem Systemausfall
  - sehr hoher Aufwand: nicht nur Platten, sondern ganze Systeme werden dupliziert

3.7 Transaktionen in verteilten Datenbanken

- Verteilte Transaktionen
  - Verteiltes Datenbanksystem zur Verwaltung komplexer Anwendungen
  - Sei $T = \{T_1, \ldots, T_n\}$ eine komplexe Transaktion auf verteiltem DBS
  - Teiltransaktionen $T_1, \ldots, T_n$ laufen auf verschiedenen Servern $DB_1, \ldots, DB_n$
  - Server sind über ein Kommunikationsnetz miteinander verbunden.



- COMMIT der Gesamttransaktion T gelingt nur, wenn alle beteiligten T_i gelingen.
  - unabhängiges COMMIT einzelner Teiltransaktionen reicht nicht aus.
  - schlägt einzelne Teiltransaktion T_i fehl, muß gesamte TA T zurückgesetzt werden.
  - Zurücksetzen von T betrifft alle, auch erfolgreiche, Teiltransaktionen $T_1, \ldots, T_n$.

  => verteiltes atomares COMMIT nötig.

Verteiltes Zwei-Phasen-Commit (2PC, TPC)

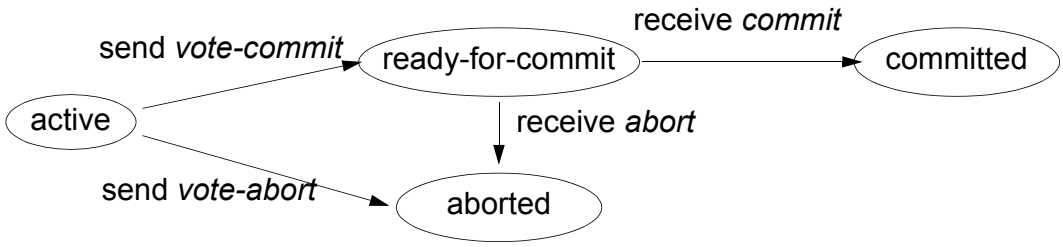
- Begriffe
  - Transaktionen $T_1, \ldots, T_n$ heißen *Teilnehmer* der Gesamttransaktion T.
  - Ein Teilnehmer hat die Rolle des *Koordinators*, der das Gesamt-COMMIT steuert.
- *Phase 1: Abstimmung der Teilnehmer*

  Die Teilnehmer $T_1, \ldots, T_n$ arbeiten unabhängig voneinander und melden am Transaktionsende ihren Zustand an den Koordinator:
  - muß ein Teilnehmer T_i zurückgesetzt werden, so kann dies unmittelbar geschehen.
    => Koordinator bekommt die Mitteilung '*vote-abort*'.
  - COMMIT am Ende einer TA T_i darf noch nicht endgültig durchgeführt werden; beide Alternativen, COMMIT und ABORT, müssen offengehalten werden.
    => Koordinator erhält die Mitteilung '*vote-commit*'.
- *Phase 2: Entscheidung des Koordinators*

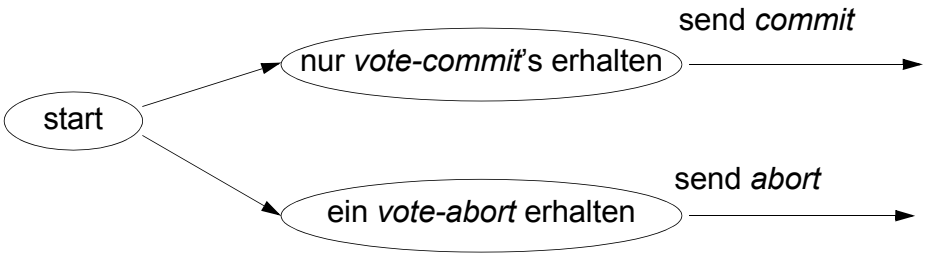
  Koordinator erhält die Ergebnisse aller Teilnehmer und fällt globale Entscheidung:
  - Liefern alle Teilnehmer '*vote-commit*', so entscheidet der Koordinator '*commit*'.
  - Liefert (mindestens) ein Teilnehmer '*vote-abort*', so entscheidet er '*abort*'.
  - Die Entscheidung wird in das Logfile des Koordinators eingetragen und allen Teilnehmern mitgeteilt.
  - Jeder Teilnehmer muß die Entscheidung des Koordinators lokal umsetzen.

Zustandsübergangsdiagramme

Teilnehmer:



Koordinator:



Zusätzliches Problem, z.B. bei Netzfehler:

Problem: Was tun bei '*ready-for-commit*', wenn T_i lange nichts vom Koordinator hört?

- Eigenmächtige Entscheidung von T_i auf *commit* oder *abort* ist nicht möglich.
- Lösung: Anderen Teilnehmer T_j nach seinem Status fragen (*HELP-ME*):
  — '*committed*':
  globale Entscheidung muß '*commit*' sein; Mitteilung an T_i ist verlorengegangen.
  => T_i darf (und muß) zu '*committed*' übergehen.
  — '*aborted*':
  Koordinator muß (mußte) unabhängig von anderen TAs auf '*abort*' entscheiden.
  => T_i kann (und muß) zu '*aborted*' übergehen.
  — '*ready-for-commit*':
  => Situation kann (noch) nicht geklärt werden.
  — '*active*', d.h. T_j hat noch nicht gewählt:
  => Situation ist noch nicht klar; T_j kann aber nach Gutdünken (z.B. wegen *help-me* Anfrage) '*vote-abort*' an Koordinator melden; dann muß T_i zu '*aborted*' übergehen.
- Umgekehrt: Falls Koordinator lange nichts von einem Teilnehmer hört (*TIMEOUT*), darf er global auf '*abort*' entscheiden.
- Beispiel Eheschließung: Brautleute sind Teilnehmer, Standesbeamter ist Koordinator.