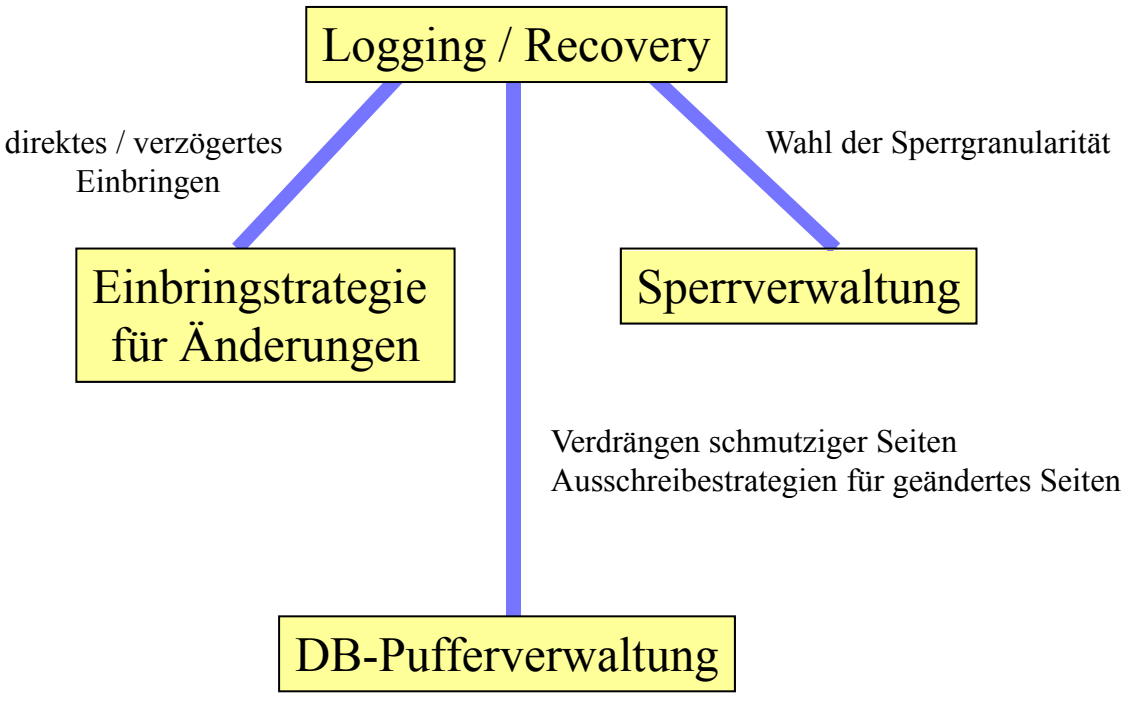
4 Recovery

Übersicht

4.3 Abhängigkeiten zu anderen Systemkomponenten

Abhängigkeiten: Übersicht



Einbringstrategien

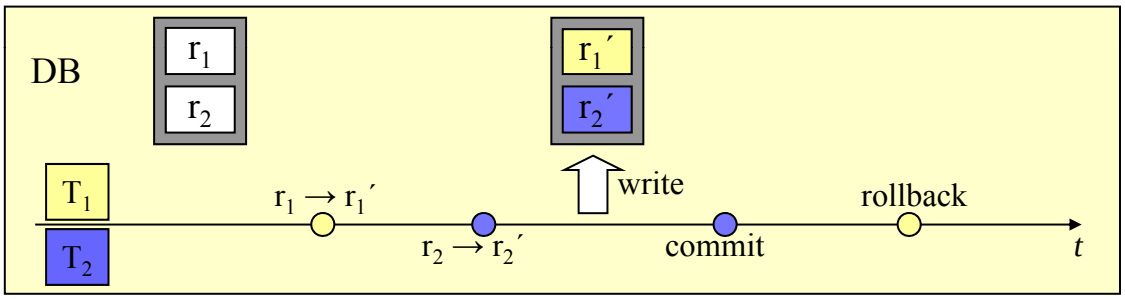
- **Direktes Einbringen (*NonAtomic, Update-in-Place*)**
  - Geänderte Objekte werden immer auf denselben Block auf Platte zurück geschrieben
  - Schreiben ist dadurch gleichzeitig Einbringen in die permanente DB
  - Es ist nicht möglich mehrere Seiten *atomar* einzubringen, d.h. Unterbrechungsfreiheit des Einbringens kann nicht garantiert werden (*NonAtomic*).

Einbringstrategien (cont.)

- **Indirektes (verzögertes) Einbringen (*Atomic*)**
  - Ziel: Einbringen in die Datenbank wird unterbrechungsfrei durchgeführt
  - Geänderte Seite wird in separaten Block auf Platte geschrieben (z.B. *Schattenspeichertechnik* oder *Twin-Block*-Verfahren)
  - Einbringen in die DB kann von *COMMIT* losgelöst werden und z.B. erst beim nächsten Sicherungspunkt stattfinden (*verzögertes* Einbringen)
  - Atomares Einbringen mehrerer geänderter Seiten ist durch Umschalten von Seitentabellen möglich (*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. Satzsperren erzwingen feine Log-Granulate
- Beispiel für Problem bei “Satzsperren 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 —> r_2’ verloren
- Lost Update, d.h. Verstoß gegen die Dauerhaftigkeit des *COMMIT*

Pufferverwaltung

- **Verdrängungsstrategien**

  Ersetzung „schmutziger“ Seiten im Puffer, d.h. $Seite_{Puffer} \neq Seite_{DB}$

  - **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
  - **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 (cont.)

- **Ausschreibestrategien** (*EOT*-Behandlung)
  - **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 Sperren 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 über mehrere TAs hinweg können gesammelt werden

Pufferverwaltung (cont.)

- Kombination der Verdrängungs- und Ausschreibestrategien

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

Pufferverwaltung (cont.)

- 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

Grundregeln zum korrekten Wiederanlauf im Fehlerfall:

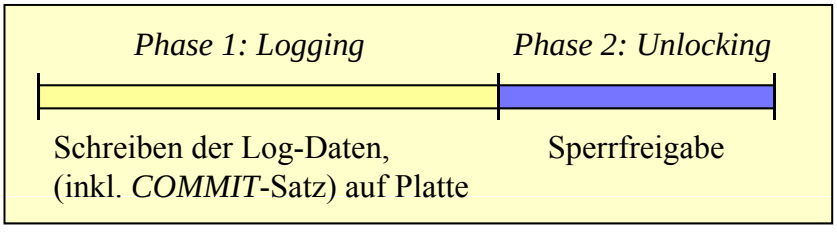
- **WAL-Prinzip (*Write-Ahead-Log*)**
  - *UNDO*-Information (z.B. *BFIM*) muss vor Änderung der DB im Protokoll stehen
  - Wichtig, um schmutzige Änderungen rückgängig zu machen
  - Nur relevant für *Steal*
  - Wichtig bei direktem Einbringen
- **COMMIT-Regel (*Force-Log-at-Commit*)**
  - *REDO*-Information (z.B. *AFIM*) muss vor dem *COMMIT* im Protokoll stehen
  - Vorraussetzung für Crash-Recovery bei *No-Force*
  - Erforderlich für Geräte-Recovery (auch bei *Force*)
  - Gilt für direkte und indirekte Einbringstrategien gleichermaßen

WAL-Prinzip und COMMIT-Regel (cont.)

- 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*-Satz
- *Phase 2: Unlocking*
  - Freigabe der Sperren (Sichtbarmachen der Änderungen)
  - Bestätigung des *COMMIT* an das Anwendungsprogramm

COMMIT-Verarbeitung (cont.)

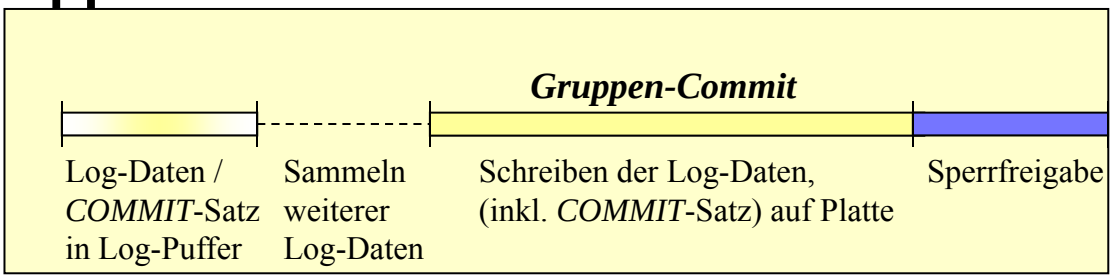
- **Problem des Standard Zwei-Phasen-Commit**

  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

COMMIT-Verarbeitung (cont.)

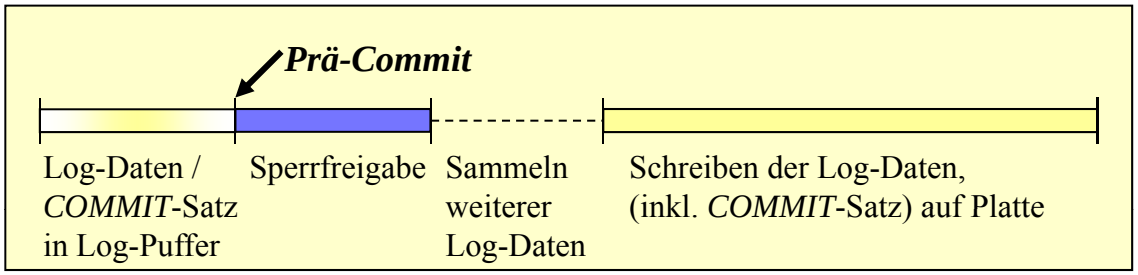
- **Gruppen-Commit**



- 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 Transaktionsraten möglich
- Nachteil: längere Sperrdauer führt zu längeren Antwortzeiten
- wird von zahlreichen DBS unterstützt

COMMIT-Verarbeitung (cont.)

- **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

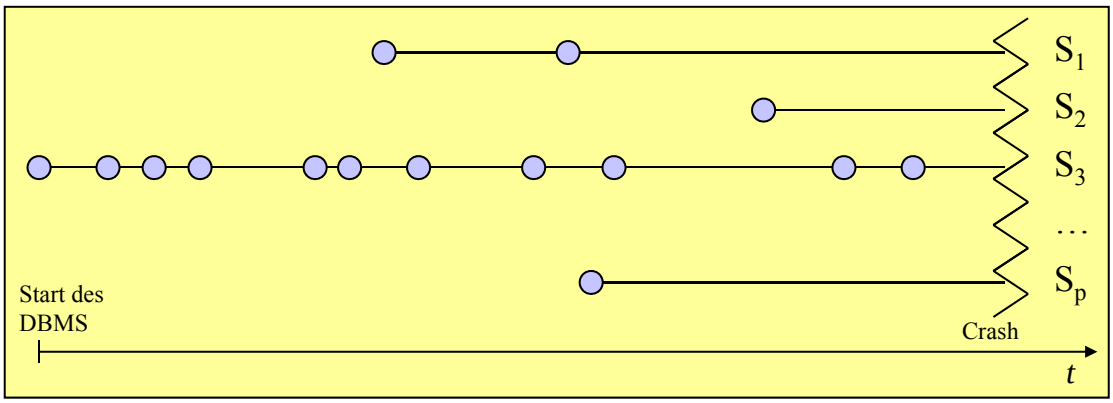
4 Recovery

Übersicht

4.4 Sicherungspunkte

Sicherungspunkte

- 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



4.4 Sicherungspunkte

Sicherungspunkte (cont.)

- **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

4.4 Sicherungspunkte

Direkte Sicherungspunkte

- **Charakterisierung**
  - Alle geänderten Seiten werden 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**)

4.4 Sicherungspunkte

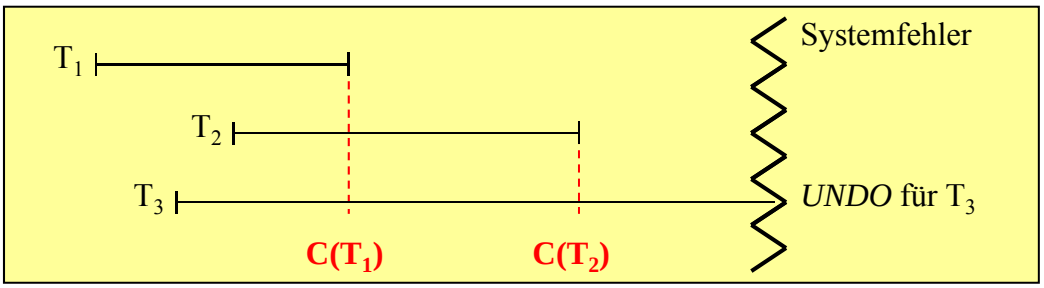
TOC: TA-orientierte Sicherungspunkte

- **TOC = *Force***, d.h. Ausschreiben aller Änderungen beim *COMMIT*
- Nicht alle Seiten im Puffer werden geschrieben, sondern nur Änderungen der jeweiligen TA
- Sicherungspunkt bezieht sich immer auf genau eine TA
- ***UNDO*-Recovery**
  Bei *Update-in-Place* ist *UNDO* nötig (*Force* → *Steal),*
  UNDO beginnt dann beim letzten Sicherungspunkt
- ***REDO*-Recovery** nicht nötig

4.4 Sicherungspunkte

TOC: TA-orientierte Sicherungspunkte (cont.)

- **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 T_1 und T_2, deshalb kein *REDO* nötig



4.4 Sicherungspunkte

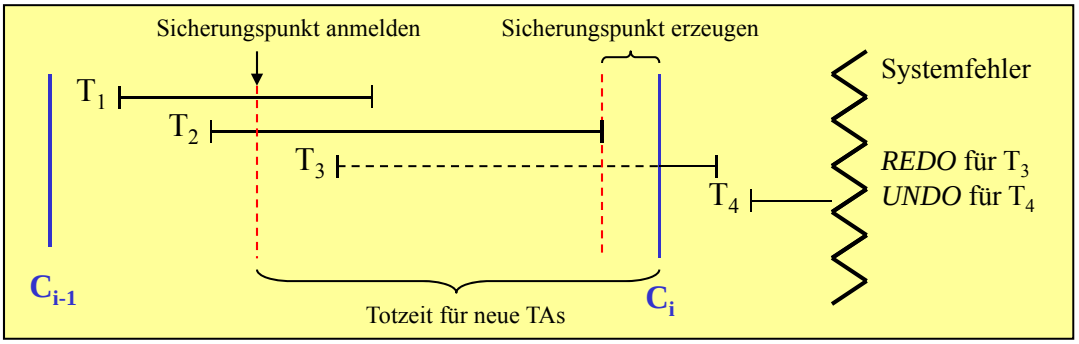
TCC: TA-konsistente Sicherungspunkte

- 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

4.4 Sicherungspunkte

TCC: TA-konsistente Sicherungspunkte (cont.)

- **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:**



4.4 Sicherungspunkte

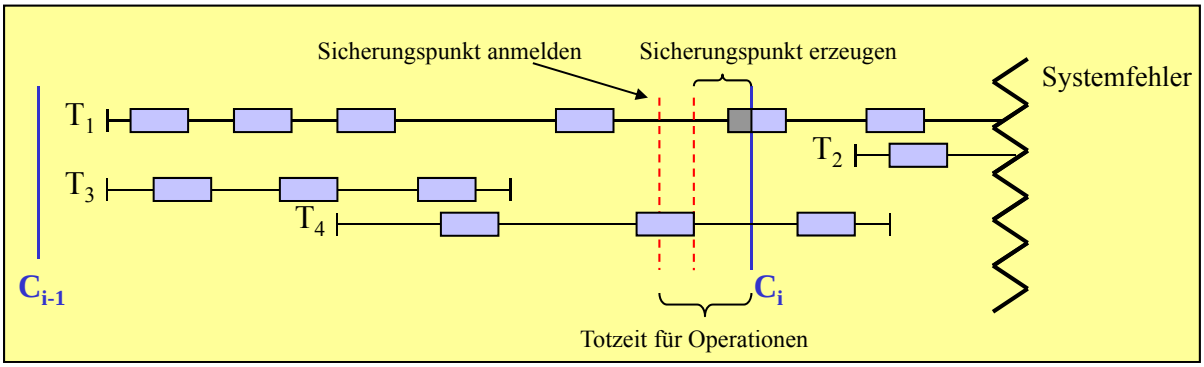
ACC: Aktions-konsistente Sicherungspunkte

- 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

4.4 Sicherungspunkte

ACC: Aktions-konsistente Sicherungspunkte

- **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:**



4.4 Sicherungspunkte

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

4.4 Sicherungspunkte

Indirekte Sicherungspunkte (cont.)

- **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

4.4 Sicherungspunkte

Indirekte Sicherungspunkte (cont.)

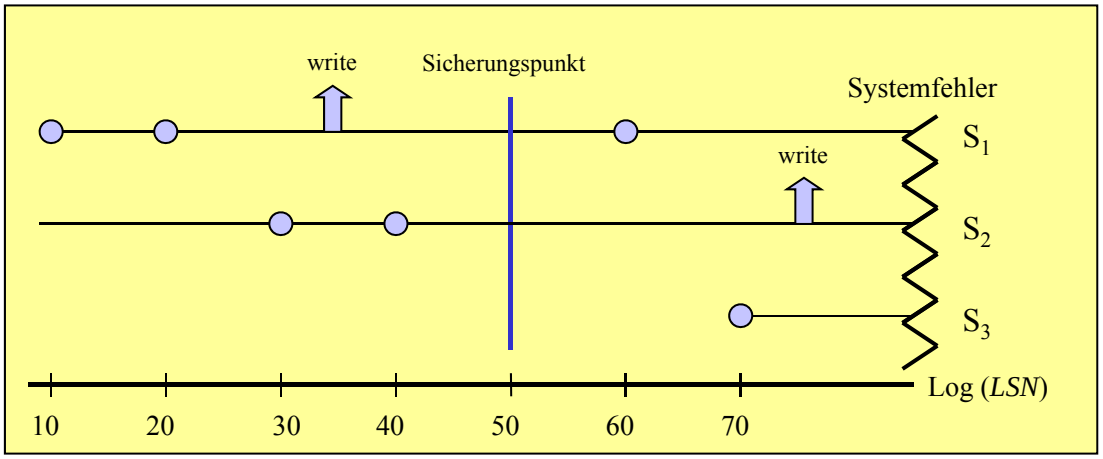
- ***UNDO*-Recovery** beginnt bei *MinLSN (siehe ACC)*

- ***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*)

4.4 Sicherungspunkte

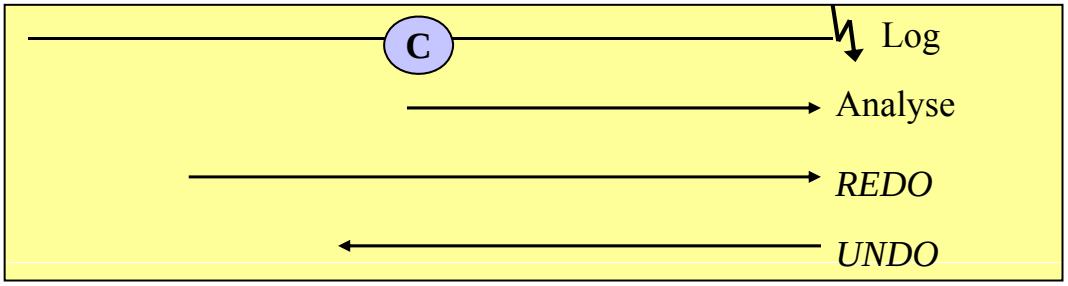
Indirekte Sicherungspunkte (cont.)

- **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



4.4 Sicherungspunkte

Allgemeine Prozedur der Crash-Recovery

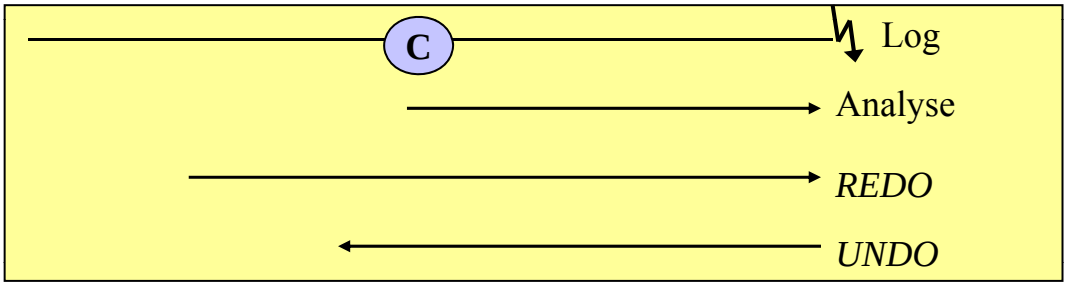


1. Analyse-Phase

- 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

4.4 Sicherungspunkte

Allgemeine Prozedur der Crash-Recovery

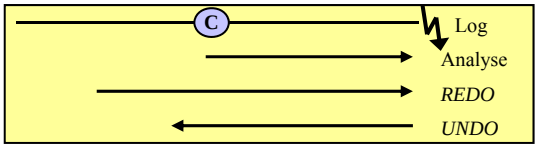


2. *REDO*-Phase

- Vorwärtslesen der Log-Datei: Startpunkt ist abhängig vom Sicherungspunktyp
- 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

4.4 Sicherungspunkte

Allgemeine Prozedur der Crash-Recovery (cont.)



3. *UNDO*-Phase

- Rückwärtslesen der Log-Datei bis zum *BOT*-Satz 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 (beendete und unbeendete)

4. Abschluß der Recovery durch einen Sicherungspunkt