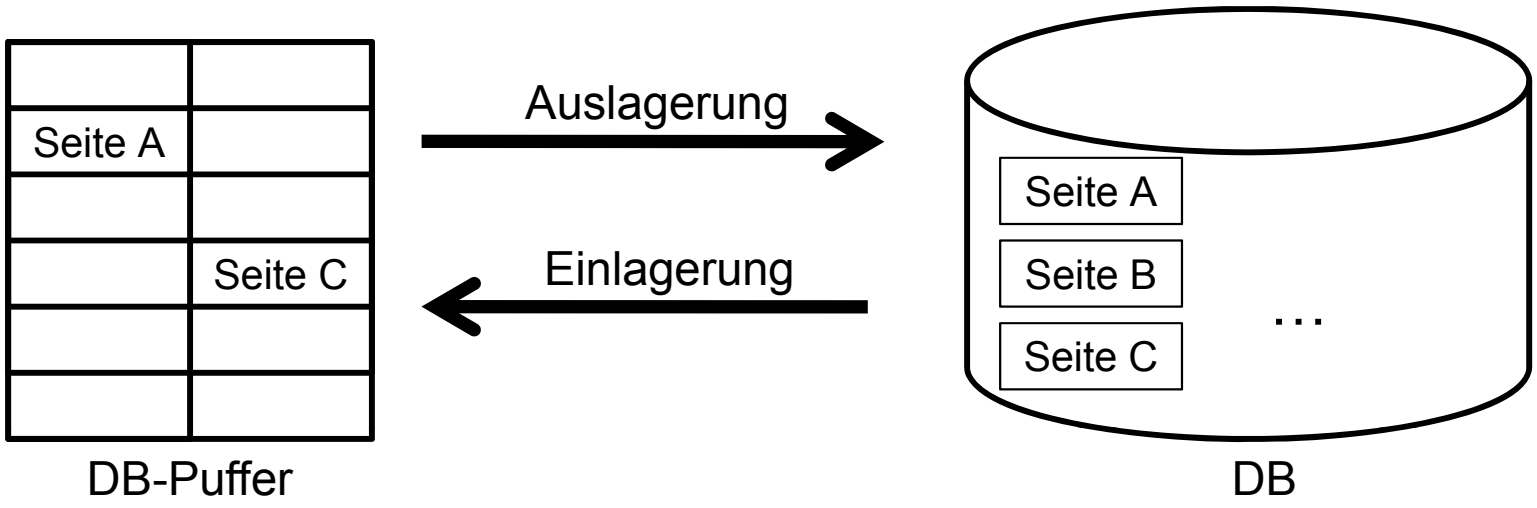
4. Recovery

4.3 Abh. zu anderen Systemkomponenten

Die Speicherhierarchie

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



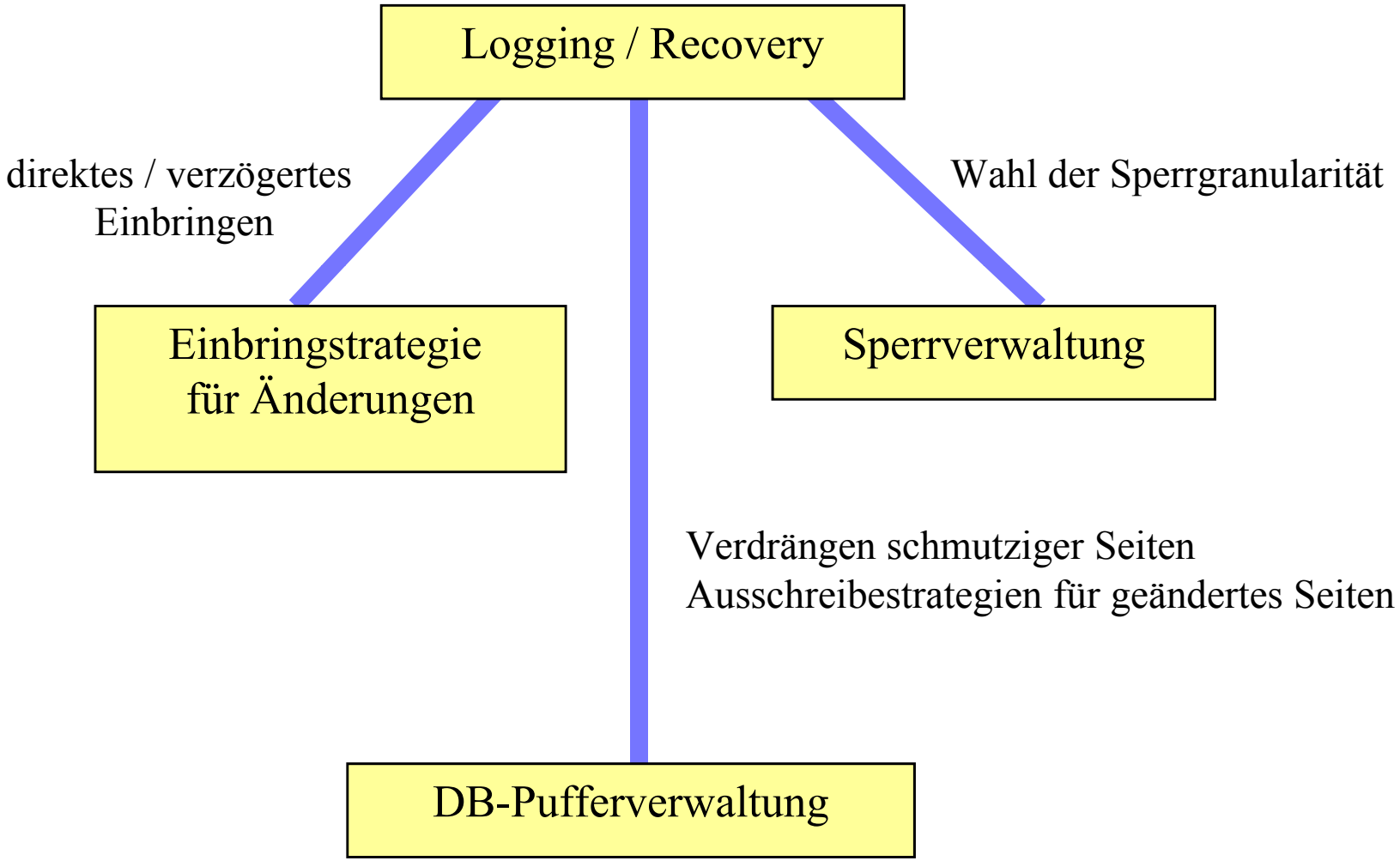
4.3 Abh. zu anderen Systemkomponenten

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***)
- Zusätzlich besteht eine Abhängigkeit der Recovery/des Loggings zur Sperrverwaltung (bei pessimistischer Synchronisation)

4.3 Abh. zu anderen Systemkomponenten

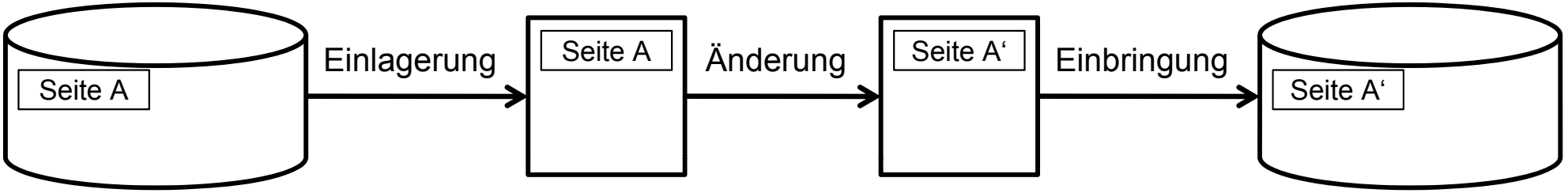
Schematischer Überblick der Abhängigkeiten



4.3 Abh. zu anderen Systemkomponenten

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

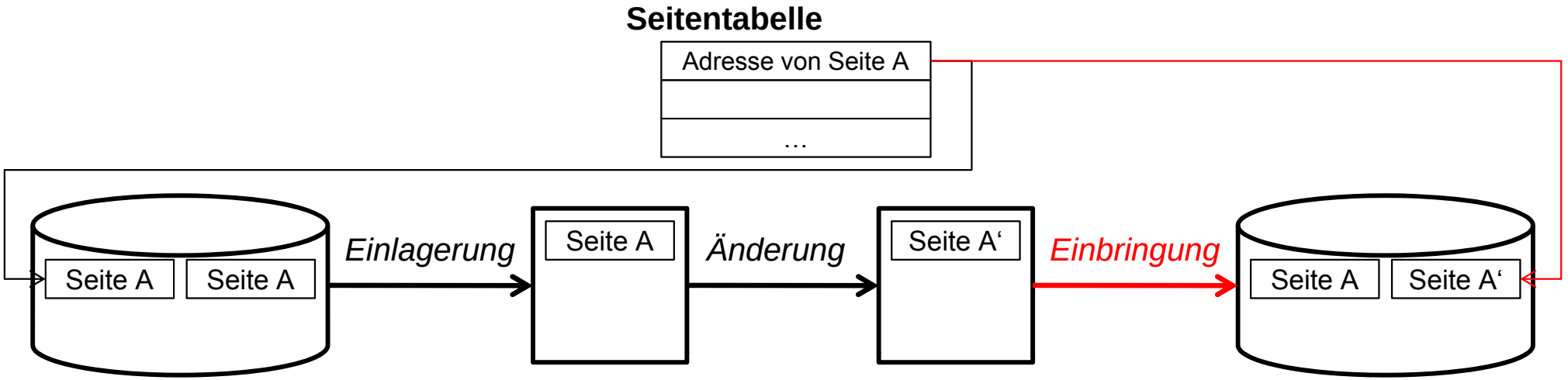
- Jede Seite hat eine Speicheradresse auf der Platte
- Geänderte Seiten werden immer auf ihren Block zurück geschrieben, d.h. der alte Inhalt der Seite in der DB wird dabei überschrieben
- Ausschreiben einer Seite ist dadurch gleichzeitig Einbringen in die permanente DB (es gibt keinen Zwischenspeicher 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



4.3 Abh. zu anderen Systemkomponenten

Einbringungstrategie: Indirektes Einbringen (Atomic)

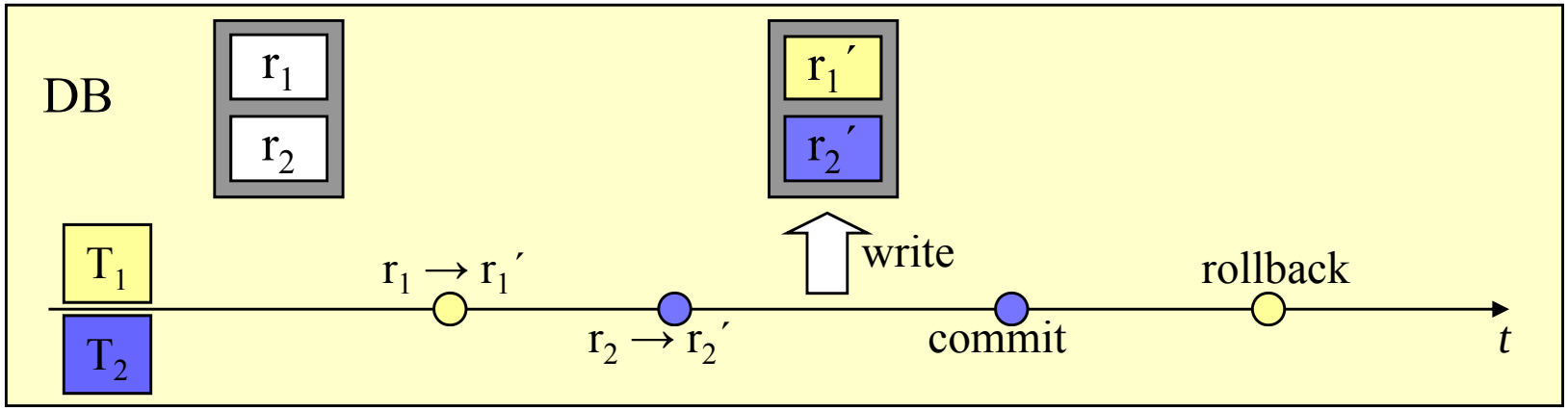
- Geänderte Seite wird in separaten Block auf Platte geschrieben
  - Twin-Block-Verfahren: jede Seite hat zwei Blöcke auf der Platte
  - Schattenspeichertechnik: nur modifizierte Seiten haben zwei Blöcke
- Atomares Einbringen mehrerer geänderter Seiten ist durch Umschalten von Seitentabellen möglich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



4.3 Abh. zu anderen Systemkomponenten

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”



- T1, T2 ändern die Datensätze r1, r2, die auf derselben DB-Seite liegen
- Die Seite wird in die DB zurück geschrieben, T2 endet mit COMMIT
- Falls T1 zurückgesetzt wird, geht auch die Änderung r2 —> r2’ 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: SeitePuffer ≠ 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 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 von mehreren TAs können gesammelt werden

4.3 Abh. zu anderen Systemkomponenten

Kombination:

	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:
    - 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**)

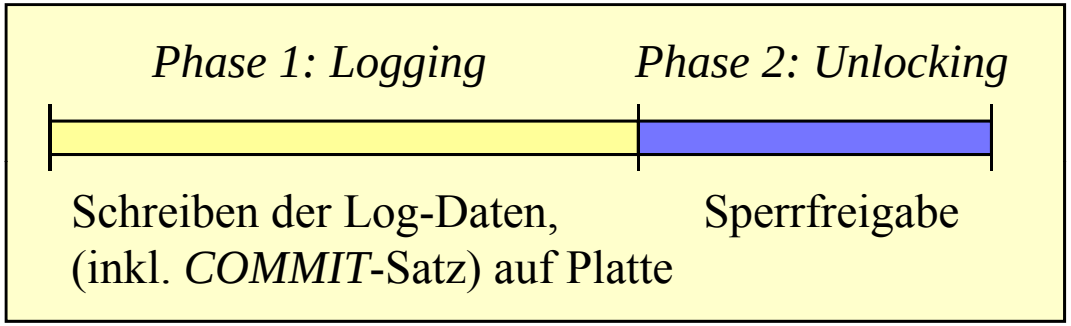
4.3 Abh. zu anderen Systemkomponenten

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
  - Vorraussetzung 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

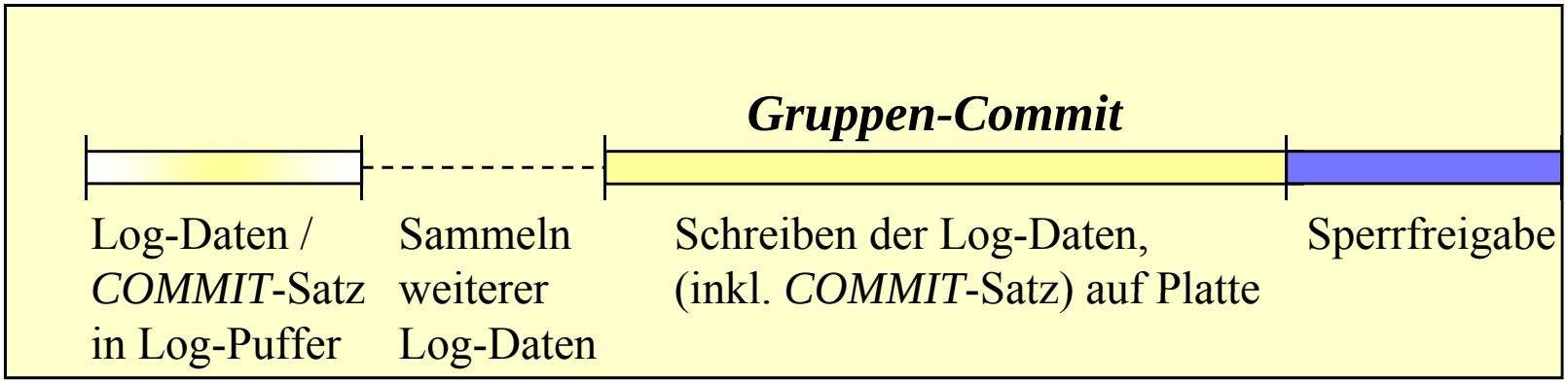
COMMIT-Verarbeitung

- ***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



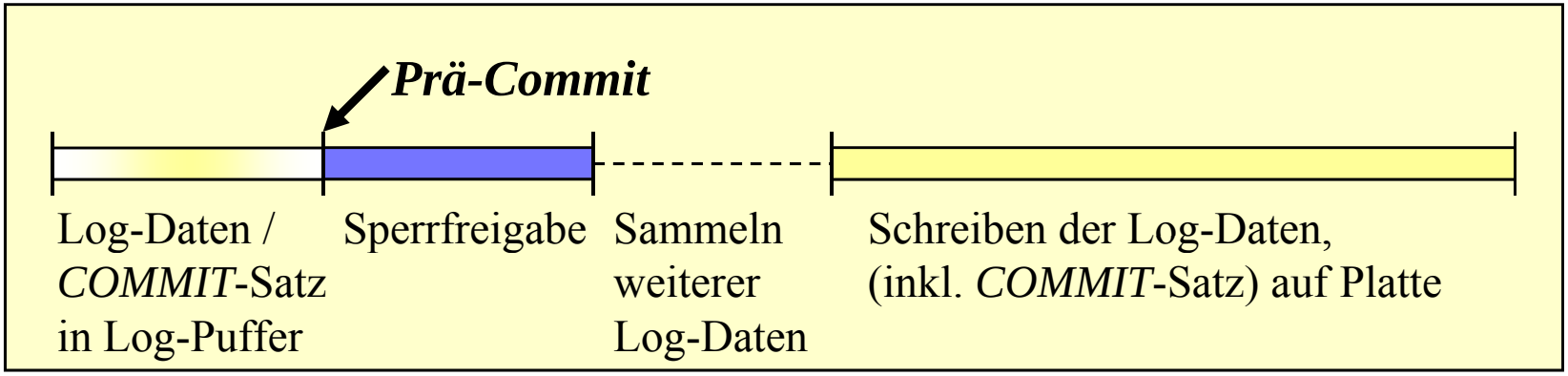
4.3 Abh. zu anderen Systemkomponenten

- ***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



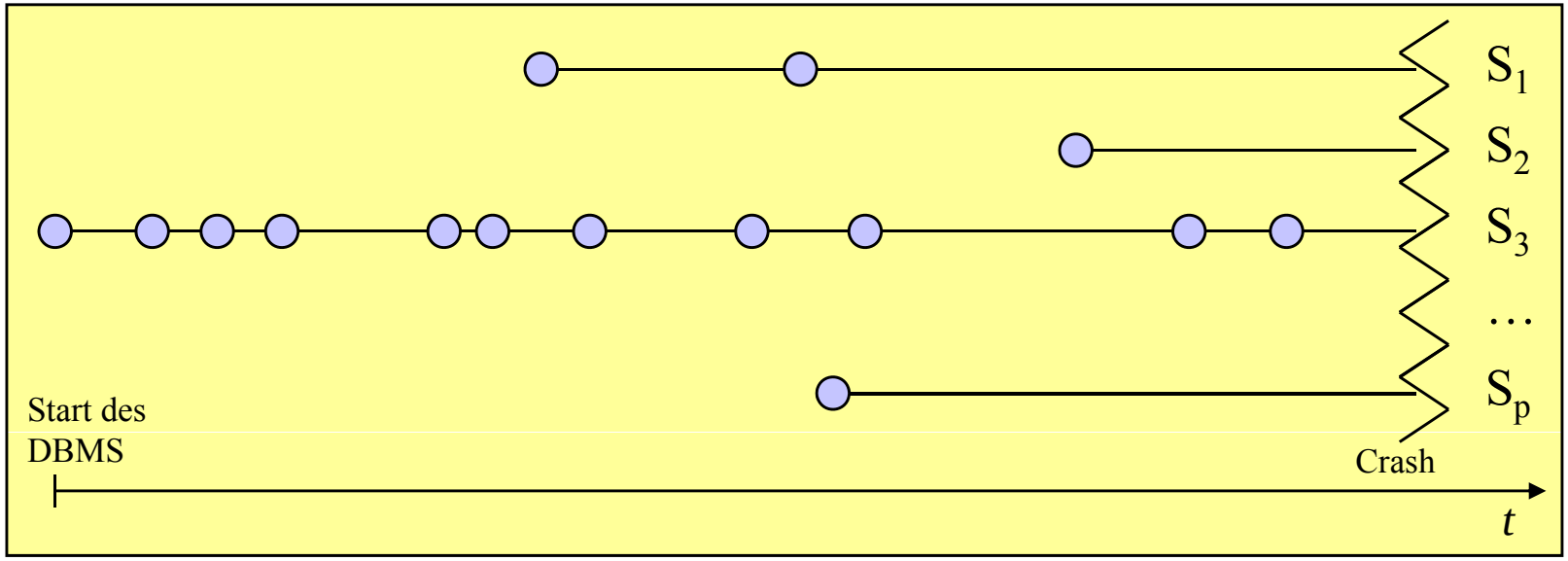
4.3 Abh. zu anderen Systemkomponenten

- ***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



4.4 Sicherungspunkte

- 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



4.4 Sicherungspunkte

- 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 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***)

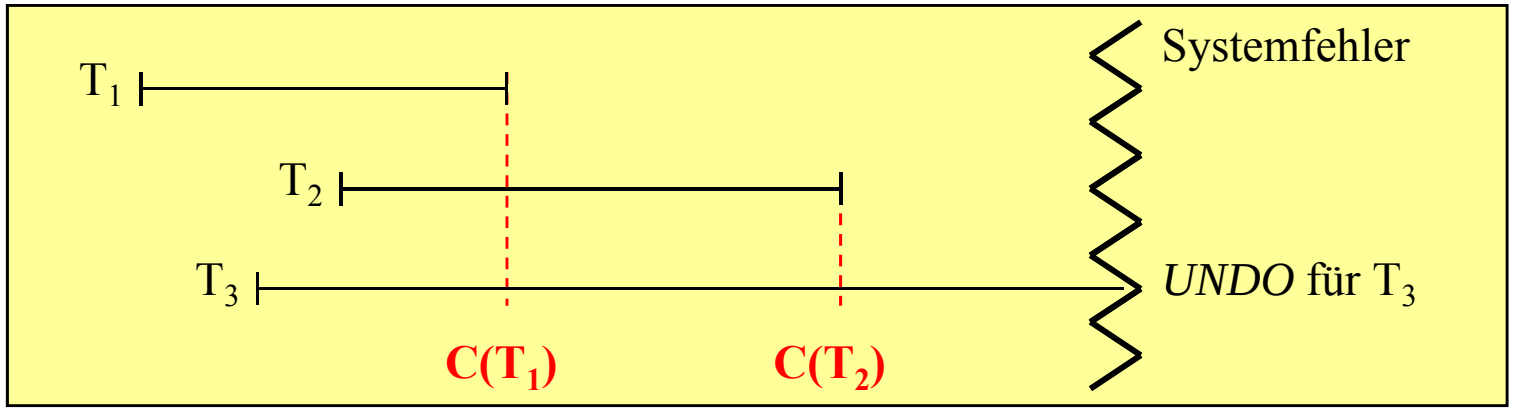
4.4 Sicherungspunkte

TOC: TA-orientierte Sicherungspunkte

- TOC entspricht *Force*, d.h. Ausschreiben aller Änderungen beim COMMIT oder anders gesagt: jedes COMMIT definiert einen Sichrungspunkt
- 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*!)

4.4 Sicherungspunkte

- 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



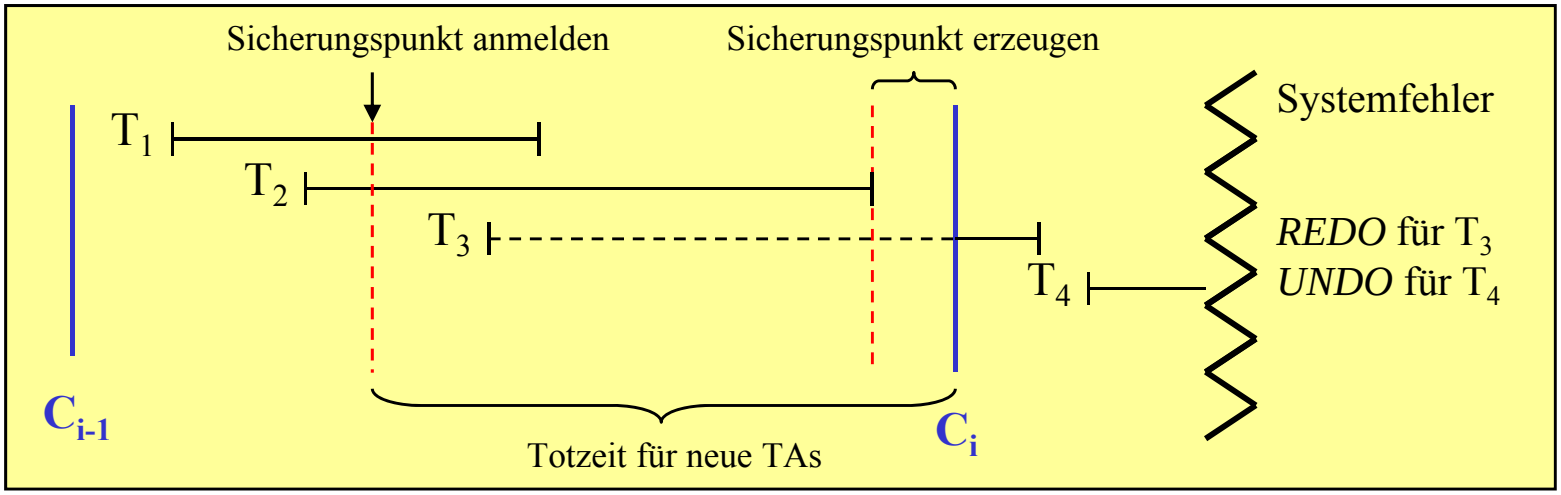
4.4 Sicherungspunkte

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

- 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:



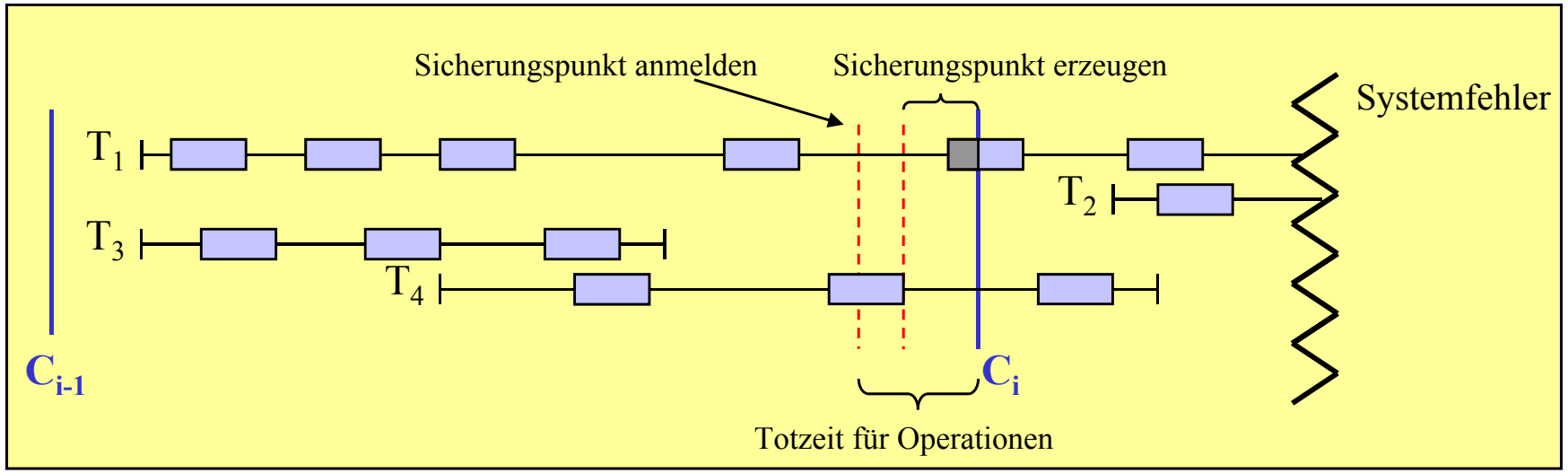
4.4 Sicherungspunkte

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

- 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

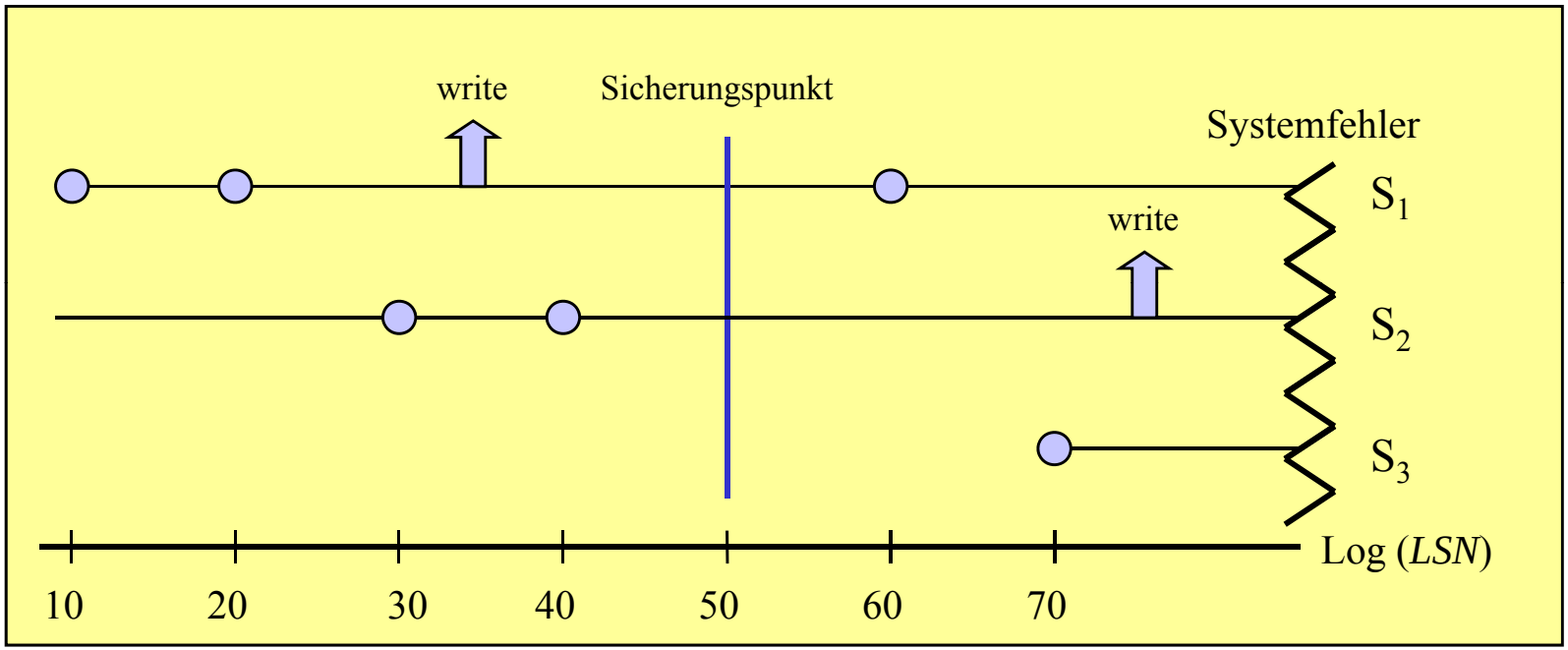
- Charakterisierung
  - Direkte Sicherungspunkte: hoher Aufwand bei großen DB-Puffern nicht akzeptabel
  - Indirekte Sicherungspunkte: Änderungen werden nicht vollständig ausgeschrieben
  - DB hat keinen Aktions- oder TA-konsistenten Zustand, sondern unscharfen (fuzzy) Zustand
- Erzeugung eines indirekten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

4.4 Sicherungspunkte

- 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*)

4.4 Sicherungspunkte

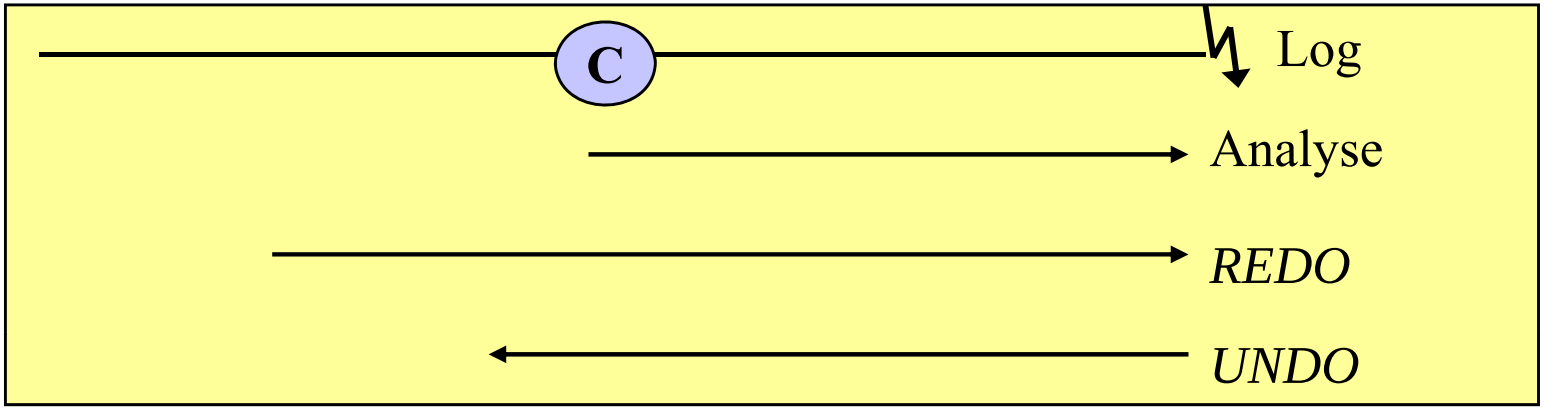
- Beispiel:



- 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

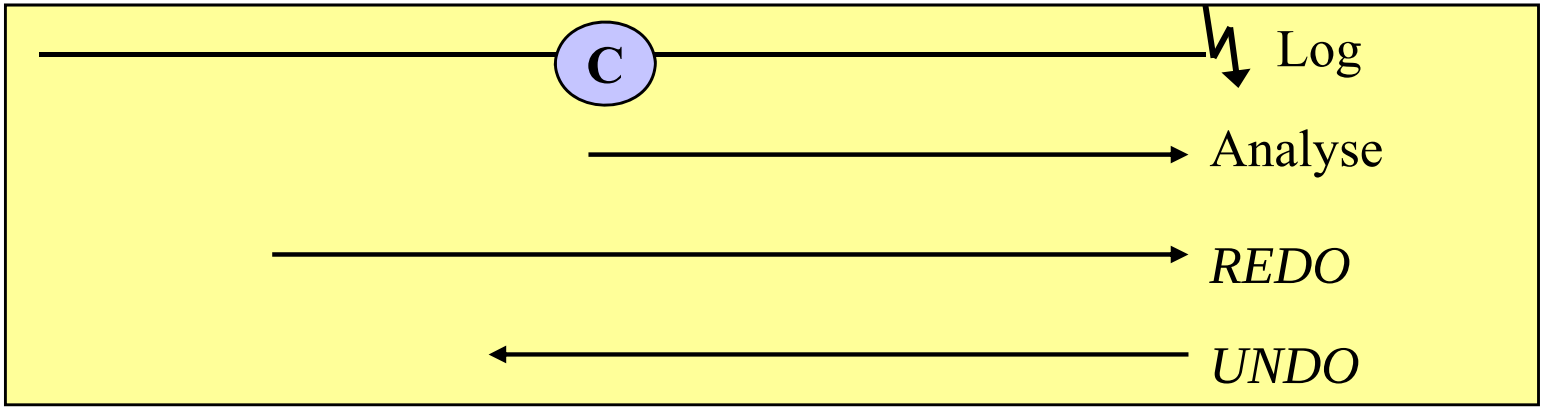
Allgemeine Prozedur der Crash-Recovery



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

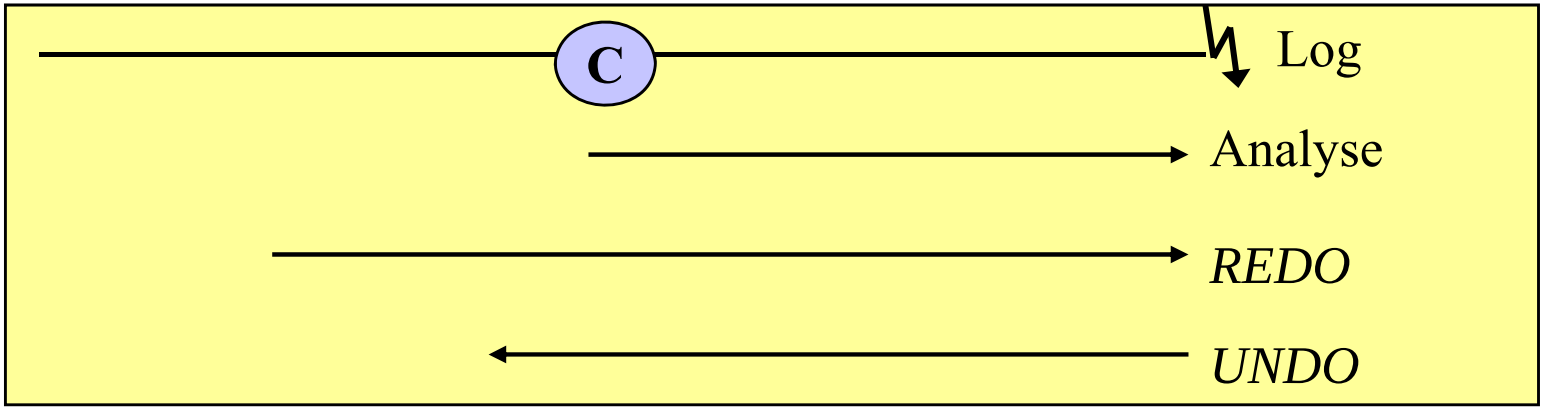
4.4 Sicherungspunkte



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



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

4.4 Sicherungspunkte

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

$$UNDO(UNDO(\ldots UNDO(A)\ldots)) = UNDO(A)$$

$$REDO(REDO(\ldots REDO(A)\ldots)) = REDO(A)$$

4.4 Sicherungspunkte

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

4.4 Sicherungspunkte

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)