Recovery

Fehlerarten: Transaktionsfehler

- Transaktionsfehler
  - Freiwilliger Transaktionsfehler durch eine ROLLBACK-Anweisung
    - Unzulässige Dateneingabe
    - Nicht erfolgreiche DB-Operation
  - Fehler im Transaktionsprogramm
    - Division durch Null
    - Adressierungsfehler
  - Systemseitiger Abbruch einer Transaktion
    - Verletzung von Integritätsbedingungen
    - Verletzung von Zugriffsbeschränkungen
  - Systemseitiger Abbruch einer oder mehrerer Transaktionen
    - Auflösung von Deadlocks (*Select a Victim*)
    - Behandlung von Systemüberlast (z.B. Sperr- oder Speicher-engpässe)

Fehlerarten (Forts.)

- Systemfehler (*Crash*)
  - Weiterer Betrieb des DBS nicht mehr möglich
    - Fehler der Hardware (z.B. Rechnerausfall)
    - Fehler der Software (z.B. Datenbank- oder Betriebssystem)
    - Umgebungsfehler (z.B. Stromausfall)
  - Verlust bzw. Verfälschung von Hauptspeicherinhalten
  - Datenbank auf Externspeicher unzerstört
- Geräte- bzw. Externspeicherfehler
  - Ausfall von Magnetplatten (*Head Crash*)
  - Rekonstruktion der durch den Ausfall verlorengegangenen Änderungen mittels Archivkopien
- Katastrophen-Recovery
  - Zerstörung eines ganzen Rechenzentrums (Verarbeitungsrechner und Externspeicher) aufgrund von Naturkatastrophen oder Anschlägen
  - Verhinderung eines Datenverlustes über verteilten Systemansatz

Rückschau - Transaktionseigenschaften

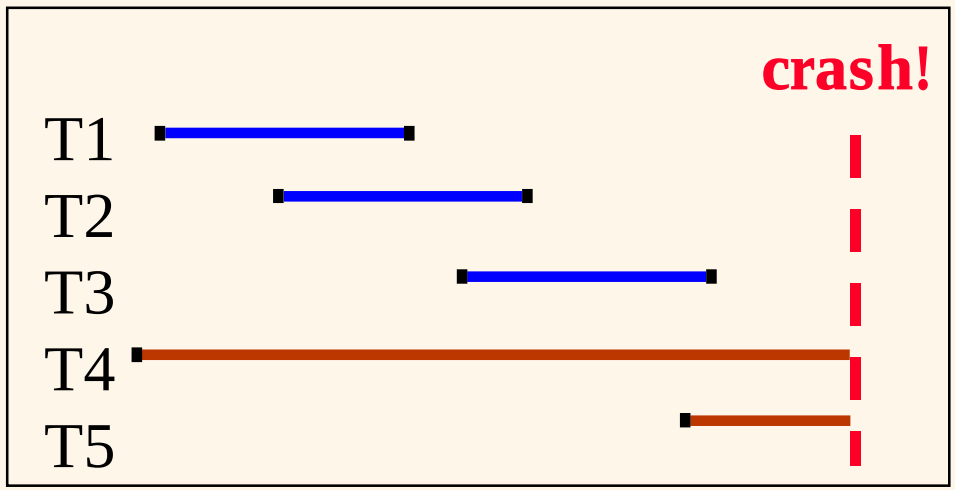
- ACID-Prinzip
  - Atomicity: Alle Aktionen einer Transaktion sind erfolgreich oder gar keine (Atomarität)
  - Consistency: Eine Transaktion garantiert den Übergang von einem konsistenten DB-Zustand zu einem anderen konsistenten DB-Zustand (Konsistenz)
  - Isolation: Die Ausführung einer Transaktion ist isoliert von der Ausführung parallel ablaufender anderer Transaktionen
  - Durability: Wenn eine Transaktion Commit macht, ist ihre Wirkung persistent (Dauerhaftigkeit)

- Recovery-Manager
  - garantiert Atomarität und Dauerhaftigkeit
  - Ist abhängig von anderen Komponenten des DBMS: Einbringstrategie, Sperrverfahren, Pufferverwaltung

Motivation

- Atomarität
  - Transaktionen können zurückgesetzt werden (Rollback)
- Dauerhaftigkeit
  - Was geschieht beim Absturz des DBMS (betrifft auch Puffer)?
- Synchronisation der Transaktionen vorausgesetzt
- Gewünschtes Verhalten nach einem Restart des Systems:
  - T1, T2 & T3 müssen dauerhaft sein
  - T4 & T5 sollten zurückgesetzt werden (keine sichtbaren Effekte hinterlassen)

Beispiel-Szenario



Einbringstrategien

- Indirekte Einbringstrategien
  - Ausschreiben geänderter DB-Seiten in separate Blöcke, so daß ursprüngliche Blöcke ungeändert bleiben
  - Ausschreiben einer Seite bringt diese noch nicht in die materialisierte Datenbank (erfordert separates Einbringen)
  - Führen von Seitentabellen, die Abbildung von Speicherseiten auf Plattenblöcke auf der Platte darstellen
  - Umschalten zwischen zwei Seitentabellen ermöglicht atomare Änderungen in der DB (Schattenspeicher-Konzept)
  - Bei Systemausfall konsistenter DB-Zustand verfügbar
  - Durch indirekte Seitenzuordnung Zerstörung der Clustereigenschaften
- Direkte Einbringstrategien (Update-in-Place)
  - Seite wird immer auf den gleichen Block zurückgeschrieben (somit direktes Überschreiben oder Löschen auf Platte)

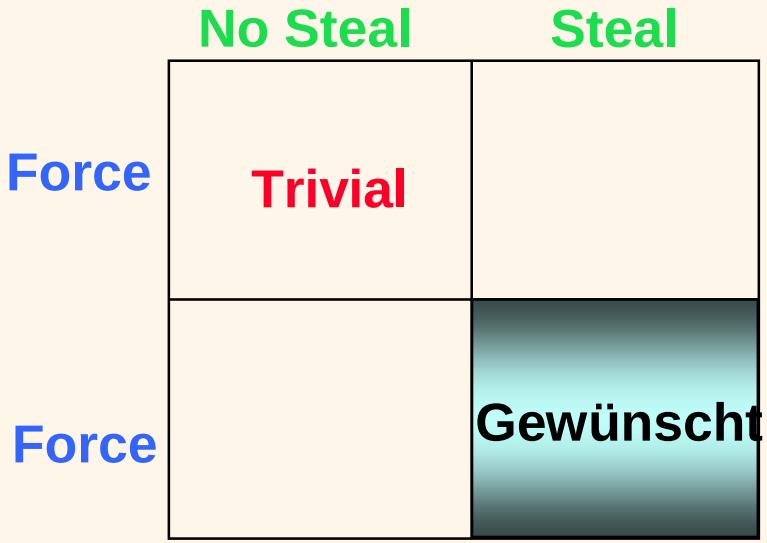
- Weitere Betrachtungen basieren auf Update-in-Place Strategie

Pufferverwaltung

- Ausschreiben schmutziger Änderungen?
  - *No Steal*: Seiten mit schmutzigen Änderungen dürfen nicht aus dem Puffer ersetzt (“gestohlen“) werden. Somit ist garantiert, daß die materialisierte Datenbank nach einem Crash keine Änderungen von nicht erfolgreich beendeten Transaktionen enthält. Keine Undo-Recovery notwendig. Blockiert für lange Änderungstransaktionen den Puffer (i.allg. mehr schmutzige Seiten als Puffergröße)
  - *Steal*: Erlaubt die Ersetzung schmutziger Seiten. Demnach ist nach einem Systemfehler eine Undo-Recovery erforderlich, um die Änderungen nicht erfolgreicher Transaktionen zurückzusetzen.
- Änderungen einer Transaktion bis zum Commit ausschreiben?
  - *Force*: Alle geänderten Seiten werden spätestens bis zum Transaktionsende (vor dem Commit) in die permanente DB geschrieben. Damit entfällt die Notwendigkeit einer Redo-Recovery nach einem Crash.
  - *No Force*: Verzichtet auf das “Hinauszwingen“ der Änderungen, stattdessen können die Seiten nach Ende der ändernden TA geschrieben werden. Erfordert Redo-Recovery nach einem Crash für die erfolgreichen Transaktionen.

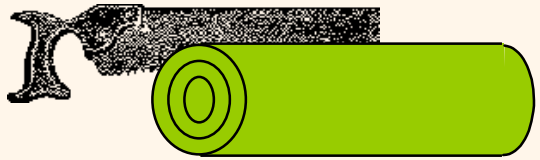
Pufferverwaltung

- Warum nicht jedes Write auf die Platte schreiben (Force)?
  - Schlechte Antwortzeit
  - Garantiert aber Persistenz
- Ersetzung von Seiten von nicht-beendeten Transaktionen (Steal)?
  - Wenn nicht, schlechter Durchsatz
  - Wenn doch, wie kann Atomariät garantiert werden?
- Günstigste Strategie:
  No Force / Steal



- Steal: Beim Zurückschreiben einer Seite P muß der alte Wert der Seite zu diesem Zeitpunkt vermerkt werden, um UNDO möglich zu machen (Atomarität)
- No Force: Schreibe so wenig wie möglich (an einem sicheren Ort) Informationen über die modifizierten Seiten, zum Commit-Zeitpunkt einer Transaktion, um REDO möglich zu machen

Idee: Logging



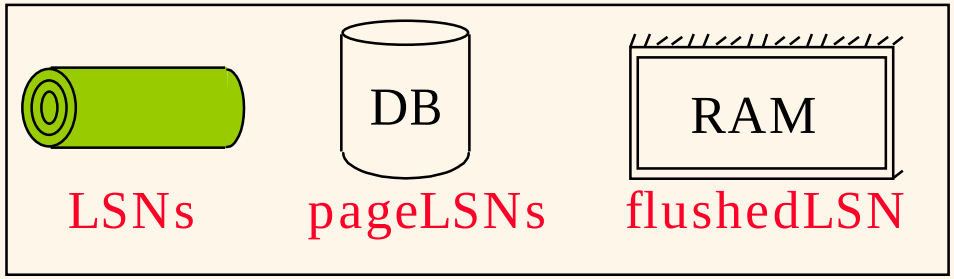
- Aufzeichnen der REDO and UNDO Information für jedes Update in einem *Log*
  - Sequentielles Schreiben auf das Log (auf einer separaten Platte)
  - Minimale Information (Differenz) ins Log schreiben, so daß mehrere Updates auf eine einzige Seite im Log passen
- Unterscheide physisches vs. logisches Protokollieren
  - Physisch: Speicherung der Log-Infos auf Ebene der physischen Objekte(z.B. Seiten)
    - Zustands-Logging: Zustände der Objekte vor und nach der Änderung (alter Zustand: Before-Image, neuer Zustand: After-Image)
    - Übergangs-Logging: Speicherung der Zustandsdifferenz
  - Logisch: Speicherung der Änderungsoperationen mit ihren Parametern (z.B. INSERT mit Attributwerten)
    - Garantiert minimalen Log-Umfang

Write-Ahead Logging (WAL)

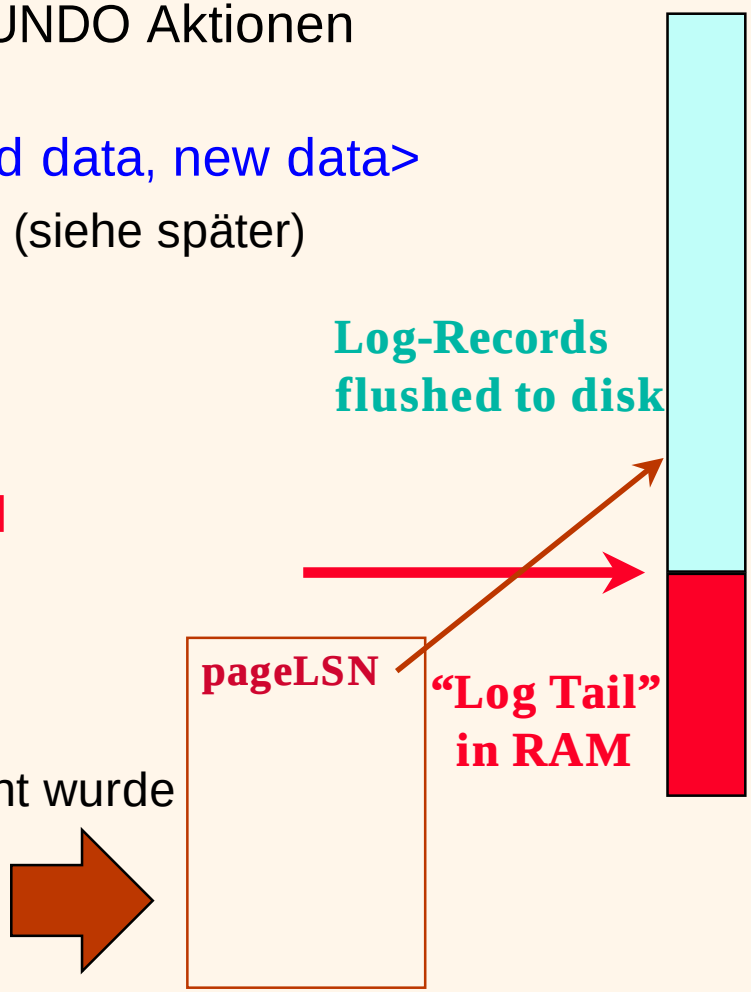
- Write-Ahead Logging Protokoll:
  - ① *Write-Ahead-Log-Prinzip*: Vor dem Schreiben einer schmutzigen Änderung in die materialisierte Datenbank muß die zugehörige Undo-Information (z.B. Before-Image) in die Log-Datei geschrieben werden (Logging *vor* Schreiben in die Datenbank). Nur für Steal relevant.
  - ② *Commit-Regel*: *Vor dem Commit* einer Transaktion sind für ihre Änderungen ausreichende Redo-Informationen (z.B. After-Images) zu sichern
- #1 garantiert Atomarität
- #2 garantiert Dauerhaftigkeit

- Beschreibung des Logging- und Recovery-Verfahrens am Beispiel des ARIES-Algorithmus (Vorbild für Recovery-Algorithmen in vielen DBMS)

WAL & Log



- Log: Eine geordnete Liste von REDO/UNDO Aktionen
  - Log-Record enthält:
    <transID, pageID, offset, length, old data, new data>
  - und zusätzliche Kontrollinformationen (siehe später)
- Jeder Log-Record hat eine eindeutige Log Sequence Number (LSN)
  - LSNs immer aufsteigend
- Jede *Datenseite* enthält eine pageLSN
  - Die LSN des jüngsten Log-Record für ein Update auf dieser Seite
- System kontrolliert die flushedLSN
  - Größte LSN, die aus dem RAM geflusht wurde
- WAL: *Vor* dem Schreiben einer Seite
  - pageLSN ≤ flushedLSN



Log-Sätze

Ein Log-Satz wird für jede der folgenden Aktionen geschrieben:

- Update
  - Nach Modifikation einer Seite wird pageLSN auf LSN des Update-Log-Record gesetzt
- Commit
  - Bei Entscheidung für Commit wird ein Commit-Record mit der Transaktions-ID geschrieben und auf Platte gezwungen (dann erst wirklich Commit der Transaktion OK)
- Abort
  - Wird bei Abort einer Transaktion geschrieben, Undo wird angestoßen
- End (bedeutet hier: Beendigung von Commit oder Abort)
  - Nach Beendigung zusätzlicher Aktionen bei Commit oder Abort
- Compensation Log Records (CLRs)
  - Beim Zurücksetzen einer Transaktion (UNDO) müssen Aktionen rückgängig gemacht werden; geschieht durch das Schreiben eines CLR

Aufbau eines Log-Satzes

- prevLSN
  - Vorherige LSN (LSNs sind verkettet)
- transID
  - ID des Transaktion, die den Log-Satz erzeugt hat
- type
  - Typ des Log-Satzes: Update, Commit, Abort, End, CLR

Für Update-Log-Records:

- pageID
  - ID der modifizierten Seite
- length
  - Länge
- offset
  - Position
- before-image
  - Alter Wert vor Änderung
- after-image
  - Neuer Wert nach Änderung

Andere Datenstrukturen beim Recovery

- Transaktions-Tabelle
  - Ein Eintrag pro aktive Transaktion
  - Enthält transID (Transaktions-ID), Status (running/commited/aborted) und lastLSN (letzte vergebene LSN)
- Dirty Page-Tabelle
  - Ein Eintrag pro schmutzige Seite im Puffer
  - Enthält recLSN - die LSN des Log-Satzes, durch den *erstmals* die Seite schmutzig gemacht wurde

Normale Ausführung einer Transaktion

- Folge von Read- und Write-Aktionen, abgeschlossen durch Commit oder Abort
  - Annahme: Write ist eine atomare Operation auf der Platte
    - Zusätzliche Dinge wären notwendig bei der Behandlung nicht-atomarer Writes
- Striktes 2PL-Protokoll
- STEAL/NO-FORCE Pufferverwaltung mit Write-Ahead Logging

Sicherungspunkte (Checkpoints)

- Sind Maßnahmen zur Begrenzung des Redo-Aufwandes nach Systemfehlern
- Redo-Recovery erforderlich für alle erfolgreich geänderten Seiten, die nur im DB-Puffer im Hauptspeicher - aber nicht in der materialisierten DB vorlagen
- Probleme bei Hot-Spot-Seiten (hohe Zugriffsrate), die nicht aus dem Puffer verdrängt werden und viele Änderungen aufweisen
  - Hoher Redo-Aufwand
  - Hoher Platzbedarf für den Log
- Direkte vs. Indirekte Checkpoints
  - Direkt: Ausschreiben aller geänderten Seiten in die mat. DB
  - Indirekt (*Fuzzy Checkpoint*): Protokollierung gewisser Status-informationen im Log
- Durchführung direkter Checkpoints sehr teuer (Abwägen zwischen Häufigkeit der Checkpoints und notwendigem Aufwand bei Redo-Recovery)

Sicherungspunkte in ARIES

- begin_checkpoint Record:
  - Zeigt den Beginn des Checkpoint an
- end_checkpoint Record:
  - Enthält aktuelle *Transaktions-Tabelle* und *Dirty Page-Tabelle*. Dies ist somit ein ‚Fuzzy Checkpoint':
    - Andere Transaktionen laufen weiter; somit stimmen diese Tabellen nur genau zu Zeitpunkt von begin_checkpoint
    - Kein Versuch, schmutzige Seiten auf Platte zu zwingen
    - Effektivität des Checkpoint begrenzt durch die älteste Änderung auf einer schmutzigen Seite (bestimmt durch *recLSN*)
    - Abhilfe: Periodisches Ausschreiben schmutziger Seiten auf Platte (Flushing), löst auch das Problem der Hot Spots
- Speichere LSN des Checkpoint-Records an einem sicheren Ort (*Master-*Record)

Big Picture: Was ist wo?

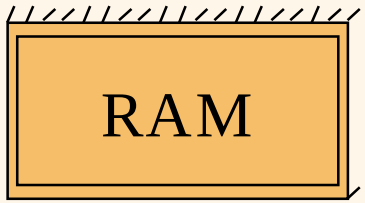
Log-Records

prevLSN
transID
type
pageID
length
offset
before-image
after-image

Datenseiten

jede
mit einer
pageLSN

Master-Record



Trans.-Tabelle

lastLSN
status

Dirty Page-Tabelle

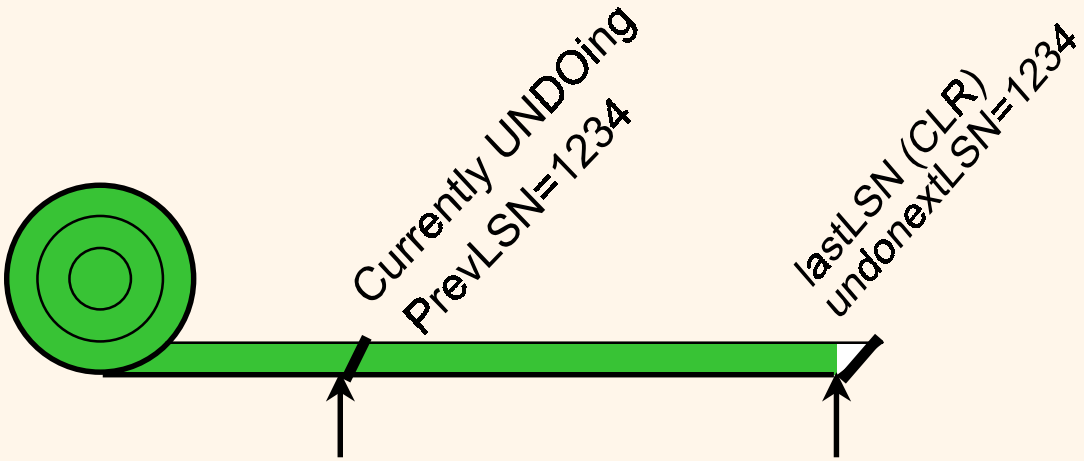
recLSN

flushedLSN

Abort einer Transaktion

- Zunächst betrachten wir den expliziten Abbruch ***einer*** Transaktion nach Transaktionsfehler (Abort)
  - Kein Systemfehler!
- "Zurückspielen" des Log in umgekehrter Richtung, Änderungen ungeschehen machen (UNDO)
  - Lese lastLSN der Transaktion aus der Transaktionstabelle
  - Zurückverfolgen der Kette der Log-Records über den prevLSN-Zeiger
  - Vor dem Start des UNDO: schreibe einen *Abort-Log-Record*
    - Wichtig für Crash-Recovery während der UNDO-Phase (siehe später)

Abort (Forts.)

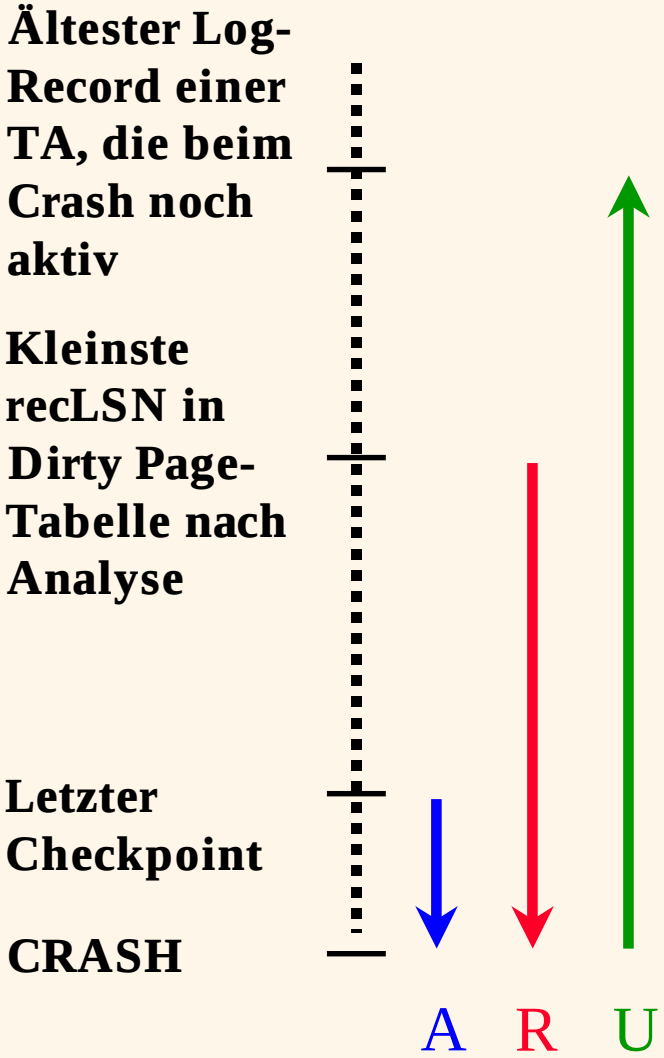


- Zur Ausführung von UNDO muß eine Sperre auf den Daten vorhanden sein
  - Kein Problem!
- Vor der Wiederherstellung des alten Wertes der Seite, schreibe einen CLR:
  - Logging wird auch während UNDO fortgesetzt!
  - CLR hat ein Extra-Feld: undonextLSN
    - Zeigt auf die nächste LSN zum undo (entspricht *prevLSN* des Satzes, der gerade rückgängig gemacht wird)
  - CLRs werden *niemals* rückgängig gemacht (höchstens REDO - garantiert Atomarität)
- Bei Ende von UNDO, schreibe einen “End” Log-Record

Commit einer Transaktion

- Schreibe Commit-Satz ins Log
- Alle Log-Sätze bis zur lastLSN der Transaktion werden auf Platte ausgeschrieben (flush)
  - Garantiert die Einhaltung der Bedingung: flushedLSN ≥ lastLSN.
  - Log Flush = sequentielles synchrones Schreiben des Log-Puffers auf Platte
  - Viele Log-Sätze pro Log-Seite
- Ende der Commit() Prozedur (z.B. Freigabe der Sperren)
- Schreibe End-Satz ins Log

Crash-Recovery: Big Picture



- Starte beim Checkpoint (ermitteln über Master-Record).
- Drei Phasen.
  - Bestimmen der Gewinner- und Verlierer-Transaktionen seit dem Checkpoint (Welche Commit? Welche Abort?) (Analyse).
  - REDO ***aller*** Aktionen

    (Wiederholung der Geschichte)
  - UNDO Effekte aller gescheiterten Transaktionen

Recovery: Analyse-Phase

- 3 Aufgaben:
  - Bestimme den Punkt im Log, an dem die REDO-Phase beginnen muß
  - Bestimme die Menge der Seiten im Puffer, die zum Crash-Zeitpunkt schmutzig waren
  - Identifiziere die Transaktionen, die zum Crash-Zeitpunkt aktiv waren und rückgängig gemacht werden müssen (UNDO)
- Rekonstruiere Zustand zum Checkpoint-Zeitpunkt
  - über end_checkpoint Satz
- Lese sequentiell vorwärts vom Checkpoint.
  - End Record: Entferne Transaktion aus Transaktionstabelle
  - Andere Records: Füge Transaktion zur Transaktionstabelle hinzu, setze lastLSN=LSN, ändere Transaktionsstatus (Commit oder Undo)
  - Update-Record: Wenn P nicht in Dirty-Page-Tabelle vorhanden:
    - Füge P zur Dirty Page-Tabelle hinzu, setze ihren recLSN=LSN.

Recovery: REDO-Phase

- Wir *wiederholen den Ablauf*, um den Zustand zum Zeitpunkt des Crash zu rekonstruieren:
  - Wiederholung *aller* Updates (auch der abgebrochenen Transaktionen!), Wiederholung der Compensation Log Records CLRs.
- Vorwärtslesen vom Log-Record, dessen LSN gleich der kleinsten recLSN in Dirty Page-Tabelle. Für jede LSN eines CLR oder Update-Log-Record, mache ein REDO der Aktion unter folgenden Voraussetzungen:
  - Betroffene Seite ist nicht in Dirty Page-Tabelle, oder
  - Betroffene Seite ist in Dirty Page-Tabelle, aber hat recLSN > LSN, oder
  - pageLSN (in DB) ≥ LSN.
- REDO einer Aktion:
  - Wiederholte Ausführung der geloggten Aktion
  - Setze pageLSN auf LSN. Kein zusätzliches Logging!

Recovery: UNDO-Phase

Zurücksetzen der Verlierer-Transaktionen in umgekehrter Reihenfolge

ToUndo={ / | / lastLSN einer Verlierer-Transaktion } aus der Tabelle der aktiven Transaktionen

Repeat:

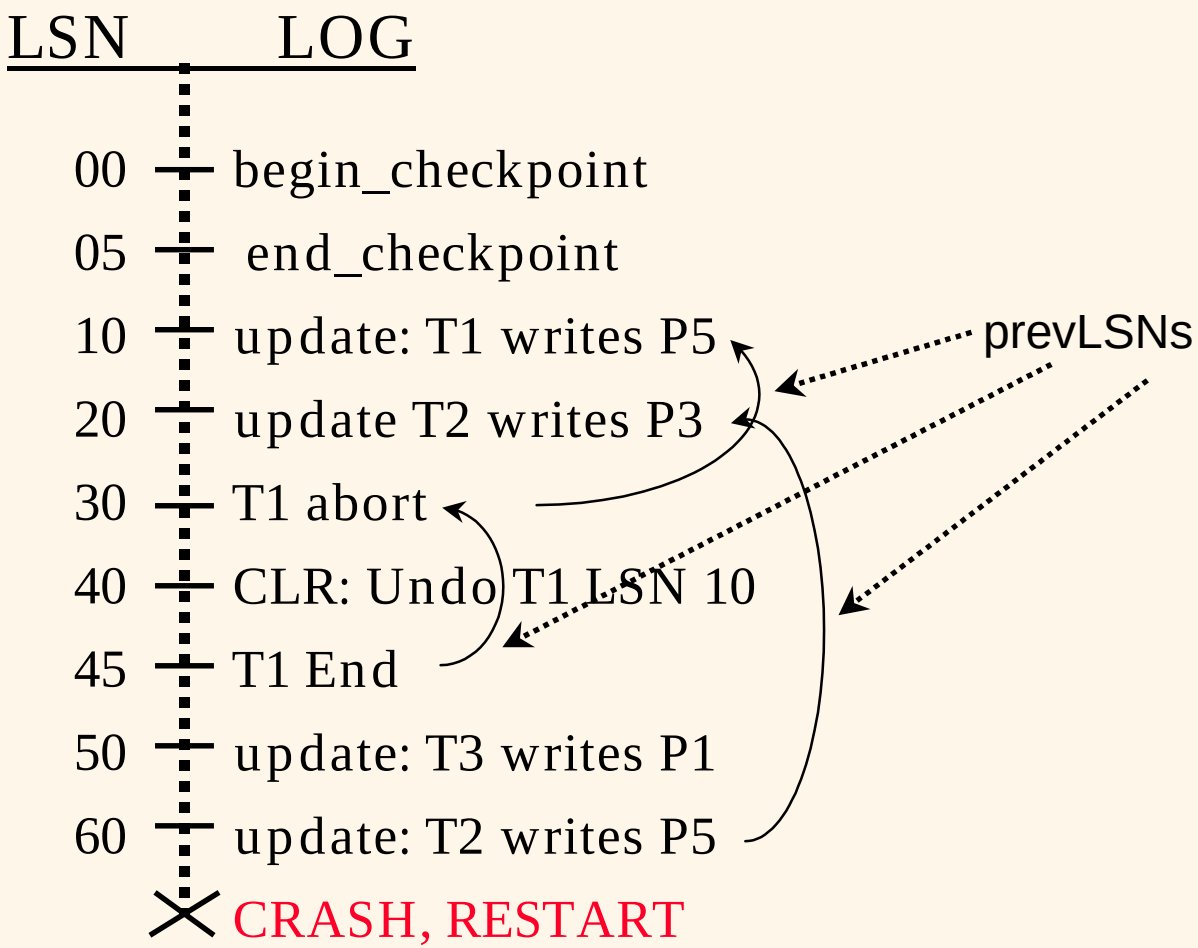
- Wähle größte LSN aus ToUndo.
- Wenn diese LSN ein CLR ist und undonextLSN==NULL
  - Schreibe einen End-Record für diese Transaktion
- Wenn diese LSN eine CLR ist, und undonextLSN != NULL
  - Füge undonextLSN zu ToUndo hinzu
- Ansonsten ist diese LSN ein Update. Rückgängigmachen des Update, schreibe einen CLR, füge prevLSN zu ToUndo hinzu

Until **ToUndo** is empty.

Beispiel Recovery

Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo



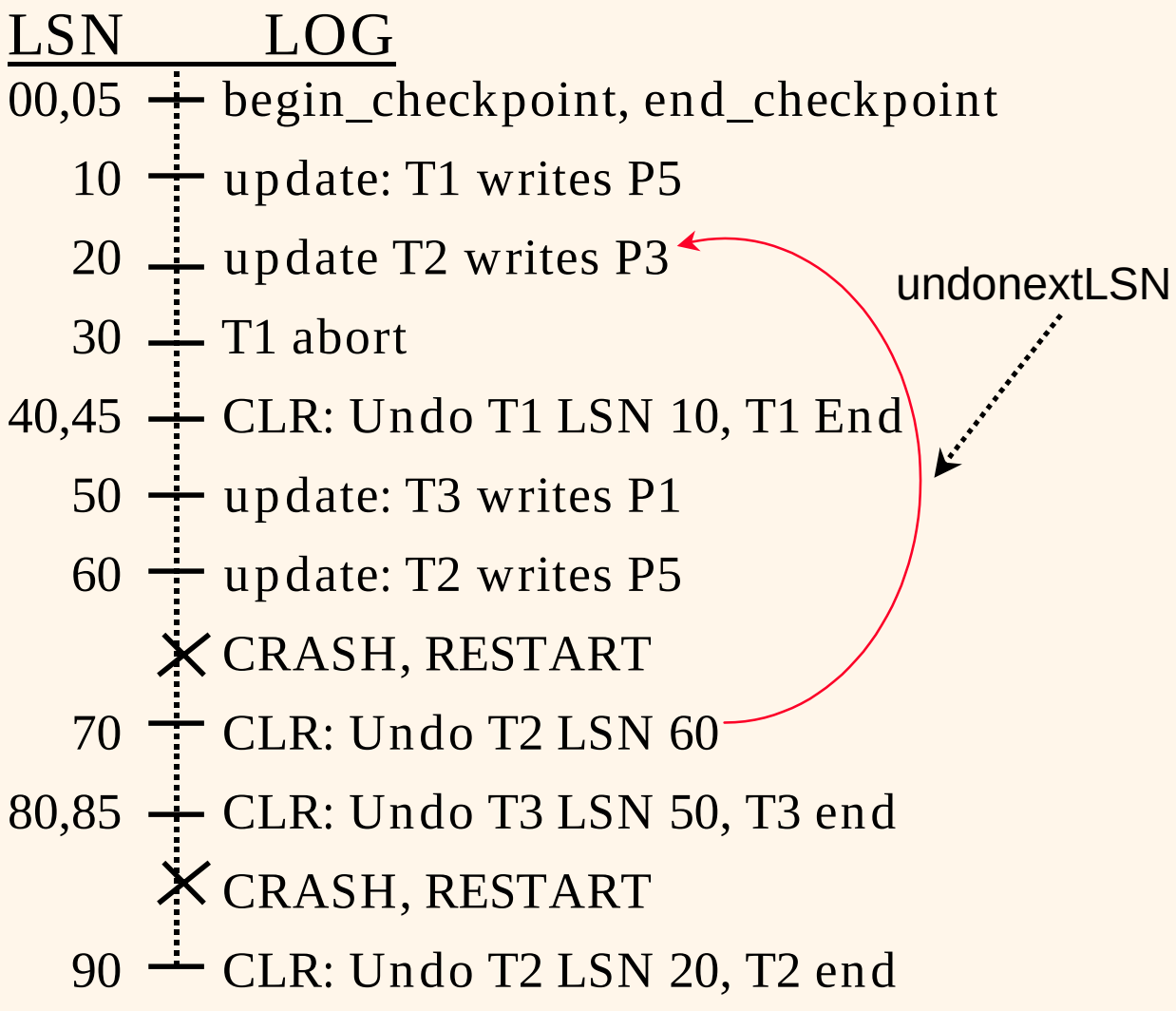
Beispiel Recovery (Forts.)

- Phase 1: Analyse
  - Bestimmung der schmutzigen Seiten:
    - P1 (mit recLSN = 50)
    - P3 (mit recLSN = 20)
    - P5 (mit recLSN = 10)
  - Bestimmung der zum Crash-Zeitpunkt aktiven Transaktionen
    - T2 (mit lastLSN = 60)
    - T3 (mit lastLSN = 50)
    - T1 bereits beendet (End-Logsatz 45)
- Phase 2: Redo
  - Wiederholung aller Aktionen ab Log-Record 10 (= min. recLSN in Dirty-Page-Tabelle)
- Phase 3: Undo
  - ToUndo = {60, 50} für T2 und T3 (jeweils größte LSN pro Transaktion)
  - Beginne mit LSN = 60: Undo Log-Record 60; Schreibe CLR 70
  - undoNextLSN = 20 (nächste Aktion für T2)
  - Weiter mit LSN = 50: Undo Log-Record 50; Schreibe CLR 80
  - undoNextLSN = NULL (T3 komplett zurückgesetzt) -> Schreibe End-Record für T3 mit LSN 85
  - usw....

Beispiel Recovery (Forts.)

Xact Table
lastLSN
status
Dirty Page Table
recLSN
flushedLSN

ToUndo



Zusammenfassung Logging / Recovery

- Recovery Manager garantiert Atomarität und Dauerhaftigkeit
- Write-Ahead-Logging (WAL), um STEAL/NO-FORCE Policy zu erlauben ohne Verzicht auf Korrektheit
- LSNs identifizieren Log-Sätze; sind rückwärtsverkettet pro Transaktione
- pageLSN erlaubt Vergleich von Datenseite und Log-Sätzen
- Checkpointing: Verfahren, um die Größe des Log zu begrenzen (verkürzt Recovery)
- Recovery arbeitet in 3 Phasen:
  - Analysis: Vorwärts vom Checkpoint.
  - Redo: Vorwärts von kleinster recLSN (ältester Log-Eintrag, der eine schmutzige Seite betrifft)
  - Undo: Rückwärts vom Ende bis zur ersten LSN der ältesten Transaktion, die zum Crash-Zeitpunkt aktiv war
- Undo heißt: Schreibe Compensation Log Record (CLR)
- Redo “wiederholt Geschichte”: Vereinfachte Logik