4. Logging und Recovery: Grundlagen

- Fehlermodell, Recovery-Arten
- Logging-Strategien
  - logisches/physisches/physiologisches und Zustands-/Übergangs-Logging
  - Seiten- vs. Eintrags-Logging
- Klassifikation von Recovery-Verfahren
  - Einbringstrategie
  - Zusammenspiel mit DBS-Pufferverwaltung
  - Commit-Behandlung, Gruppen-Commit
  - Sicherungspunkte (Checkpoints)
- Aufbau der Log-Datei, LSN-Adressierung

DB-Recovery

- automatische Behandlung aller erwarteten Fehler durch das DBVS
- Voraussetzung: Sammeln redundanter Informationen während des normalen Betriebes (Logging)
- Transaktionsparadigma verlangt:
  - Alles-oder-Nichts-Eigenschaft von Transaktionen
  - Dauerhaftigkeit erfolgreicher Änderungen
- Zielzustand nach erfolgreicher Recovery:

  Durch die Recovery-Aktionen ist der jüngste Zustand vor Erkennen des Fehlers wiederherzustellen, der allen semantischen Integritätsbedingungen entspricht, der also ein möglichst aktuelles, exaktes Bild der Miniwelt darstellt
- Forward-Recovery i.a. nicht anwendbar
  - Fehlerursache häufig falsche Programme, Eingabefehler u.ä.
  - durch Fehler unterbrochene Transaktionen sind zurückzusetzen (Backward Recovery)

Fehlerarten

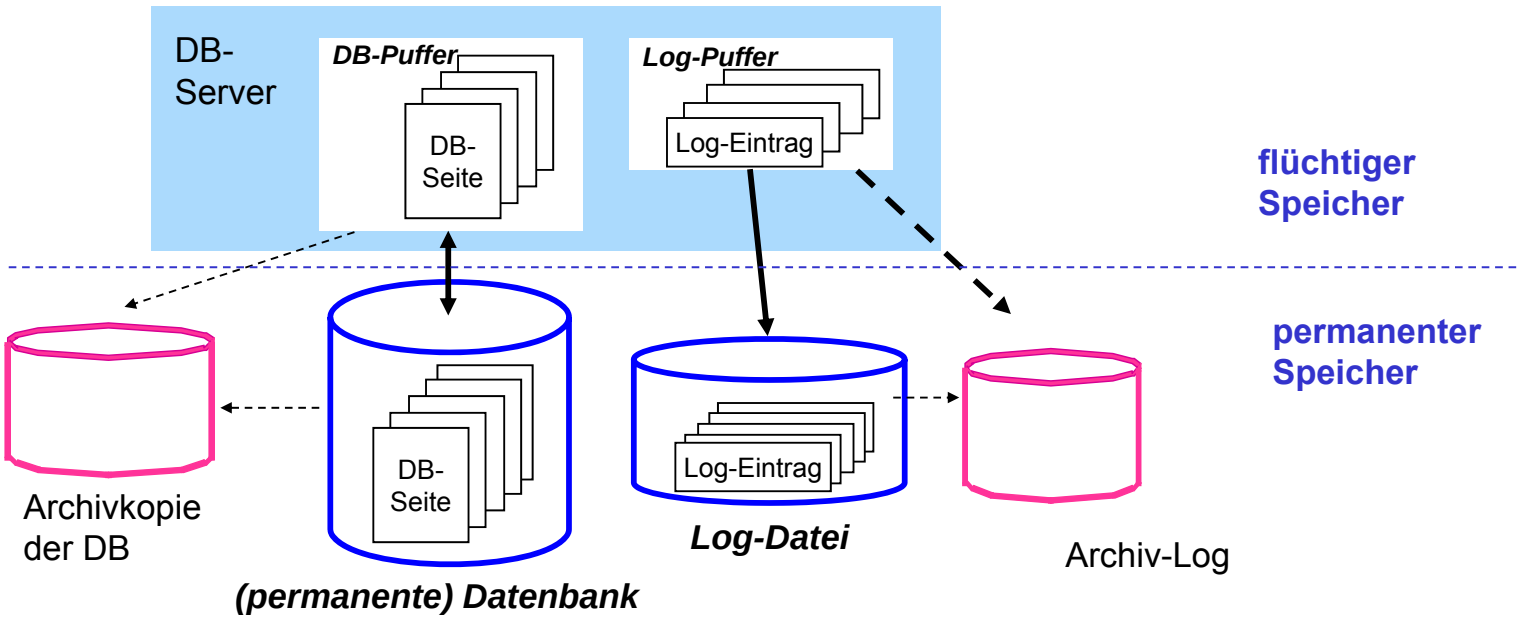
Auswirkung eines Fehlers auf	Fehlertyp	Fehlerklassifikation
eine Transaktion	- Verletzung von Systemrestriktionen • Verstoß gegen Sicherheitsbestimmungen • übermäßige Betriebsmittelanforderungen - anwendungsbedingte Fehler • z. B. falsche Operationen und Werte	*Transaktionsfehler*
mehrere Transaktionen	- geplante Systemschließung - Schwierigkeiten bei der Betriebsmittelvergabe • Überlastung des Systems • Verklemmung mehrerer Transaktionen	*Systemfehler*
alle Transaktionen (das gesamte Systemverhalten)	- Systemzusammenbruch mit Verlust der Hauptspeicherinhalte • Hardware-Fehler • falsche Werte in kritischen Tabellen - Zerstörung von Sekundärspeichern - Zerstörung des Rechenzentrums	*Systemfehler* *Gerätefehler* *Katastrophen*

Recovery-Arten

1. Zurücksetzen (Undo) einzelner Transaktionen im laufenden Betrieb (Transaktionsfehler, Deadlock, etc.)
   - vollständiges Zurücksetzen auf Transaktionsbeginn (Standard) bzw.
   - partielles Zurücksetzen auf Rücksetzpunkt (*Savepoint*) innerhalb der Transaktion
2. Crash-Recovery nach Systemfehler

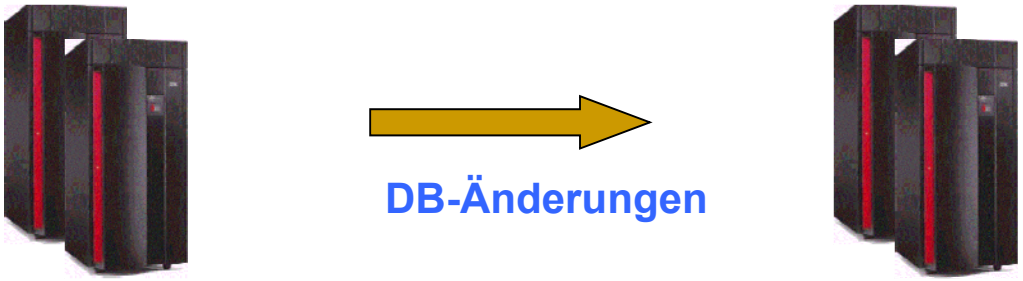
   Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustandes:
   - Redo für erfolgreiche Transaktionen (Wiederholung verloren gegangener Änderungen)
   - Undo aller durch Ausfall unterbrochenen Transaktionen (Entfernen derer Änderungen aus der permanenten DB)
3. Platten-Recovery nach Gerätefehler
   - Spiegelplatten / Disk-Arrays bzw.
   - vollständiges Wiederholen (Redo) aller Änderungen auf einer Archivkopie
4. Katastrophen-Recovery
   - stark verzögerte Fortsetzung der Verarbeitung an repariertem/neuem System mit Archivkopie (Datenverlust!) oder
   - Nutzung einer aktuellen DB-Kopie an einem geographisch separierten System

Systemkomponenten



- Pufferung von Log-Daten im Hauptspeicher (Log-Puffer)
  - Ausschreiben spätestens am Transaktionsende ("Commit")
- Log-Datei zur Behandlung von Transaktions- und Systemfehler:
  DB + Log => DB
- Behandlung von Gerätefehlern: Archivkopie + Archiv-Log => DB

DB-Mirroring für Hochverfügbarkeit

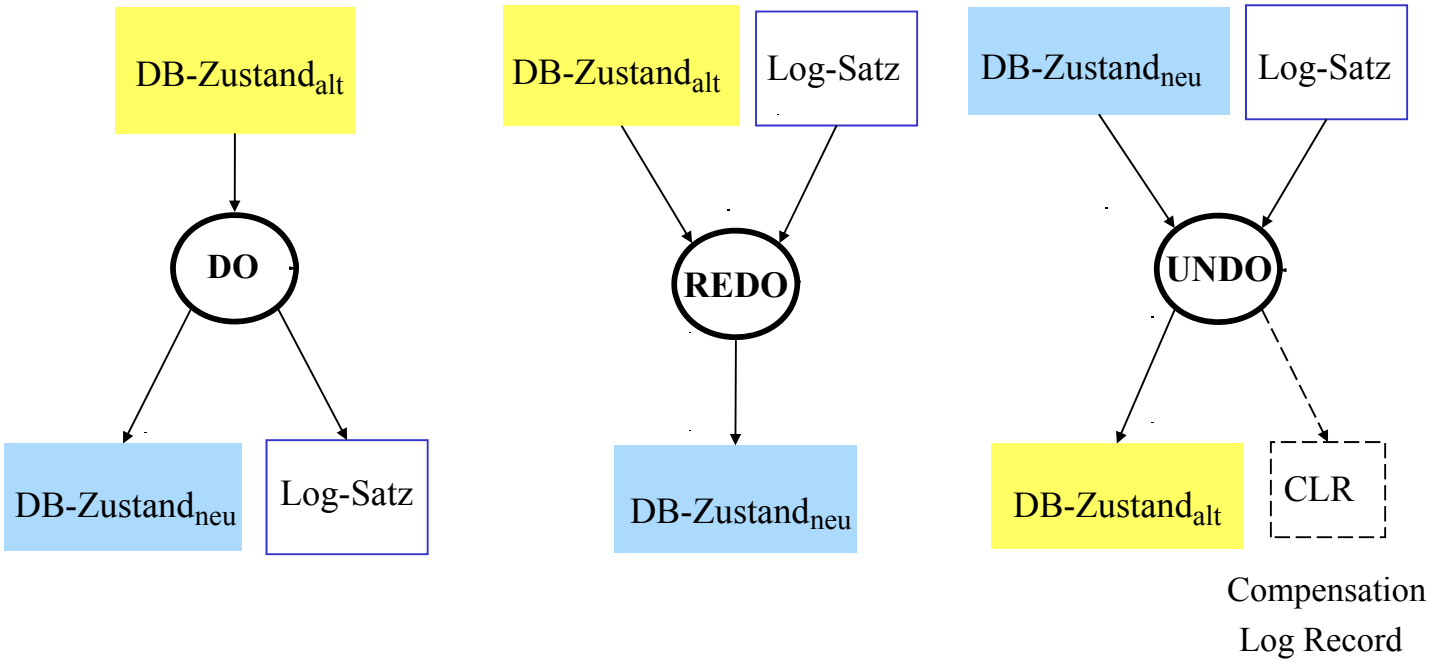


Primärsystem Sekundärsystem

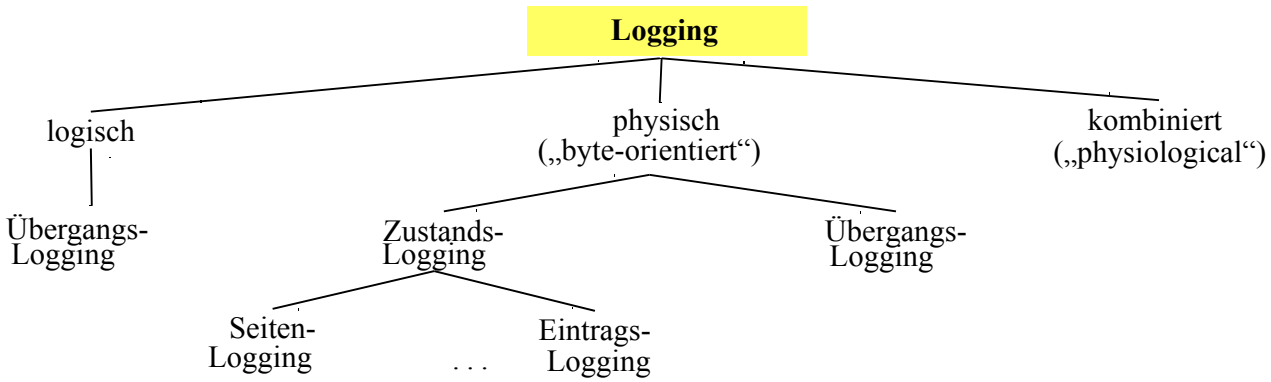
- komplette DB-Kopie an entferntem System
- fortlaufende Übertragung aller Änderungen aus Primärsystem (z.B. Log-Transfer) und Anwendung auf Kopie
- Schutz auch gegenüber Katastrophen
- synchrone oder asynchrone Aktualisierung des Sekundärsystems
- Sekundärsystem nutzbar für Lesezugriffe (Queries)

Do-Redo-Undo-Prinzip

- Logging (Protokollierung von Änderungen) im Normalbetrieb Voraussetzung für Recovery



Logging



- *Logisches Logging*
  - Protokollierung der ändernden DML-Befehle mit ihren Parametern
- *Physisches Logging*
  - Zustands-Logging: alte Zustände (*Before-Images*) und neue Zustände (*After-Images*) geänderter Objekte werden auf die Protokolldatei geschrieben
  - Übergangs-Logging: Protokollierung der Differenz zwischen Before- und After-Image

Logging: Anwendungsbeispiel

- Änderungen bezüglich einer Seite A:
  1. Ein Objekt a wird in Seite A eingefügt
  2. In A wird ein bestehendes Objekt b_{alt} nach b_{neu} geändert
- Zustandsübergänge von A: $A_1 \xrightarrow{1.} A_2 \xrightarrow{2.} A_3$

	logisch	*Physisch (Seiten-Logging)*
Zustände		Protokollierung der Before- und After-Images 1. 2.
Übergänge	Protokollierung der Operationen mit Parameter 1. 2.	Differenzen-Logging 1. 2.

- Rekonstruktion von Seiten beim Differenzen-Logging:
  - A1 als Anfangs- oder A3 als Endzustand seien verfügbar. Es gilt:

Redo-Recovery Undo-Recovery

Bewertung der Logging-Verfahren (1)

- Logisches Logging
  - Voraussetzung: nach Systemausfall müssen DML-Befehle auf der permanenten DB ausführbar sein, d.h. die DB muss nach Crash wenigstens aktionskonsistent sein
  - ist bei Update-in-Place nicht erreichbar (indirekte Seitenzuordnung erforderlich)
- physisches Logging ist bei Update-in-Place und indirekten Einbringstrategien anwendbar
- Seiten-Logging
  - einfache Recovery
  - sehr hoher Speicherbedarf und hoher Log-Aufwand, auch bei Differenzen-Logging
  - Seiten-Logging impliziert i.a. Synchronisation auf Seitenebene (zu viele Sperrkonflikte)
- Eintrags-Logging ermöglicht wesentliche Vorteile
  - geringerer Platzbedarf
  - weniger Log-E/As
  - erlaubt bessere Pufferung von Log-Daten (Gruppen-Commit)
  - unterstützt feine Synchronisationsgranulate

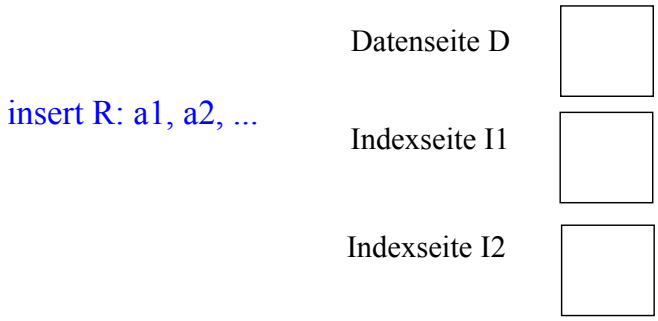
Physiologisches Logging

- einfaches, "byte-orientiertes" physisches (Eintrags-) Logging oft zu restriktiv
  - unnötig starr v.a. bezüglich Lösch- und Einfügeoperationen

- Kombination physische/logische Protokollierung: Physical-to-a-page, Logical-within-a-page
  - Protokollierung von elementaren Operationen innerhalb einer Seite
  - jeder Log-Satz bezieht sich auf 1 Seite
  - mit Update-in-Place verträglich

- Beispiel

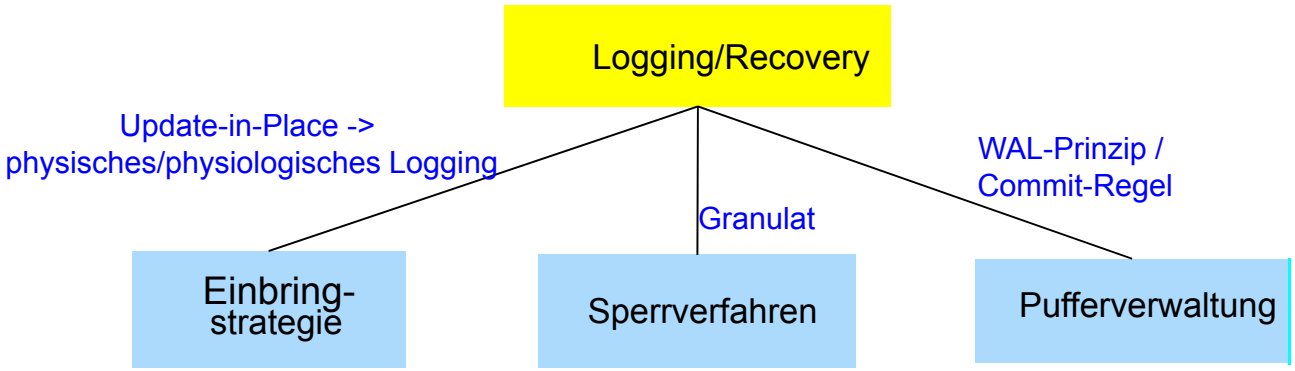
logisches Logging | *physisches Eintrags-Logging* | *physiologisches Logging*



Bewertung der Logging-Verfahren (2)

	Logging-Aufwand	Restart-Aufwand
logisches Logging		
Seiten-Logging		
Eintrags-Logging / physiologisches Logging		

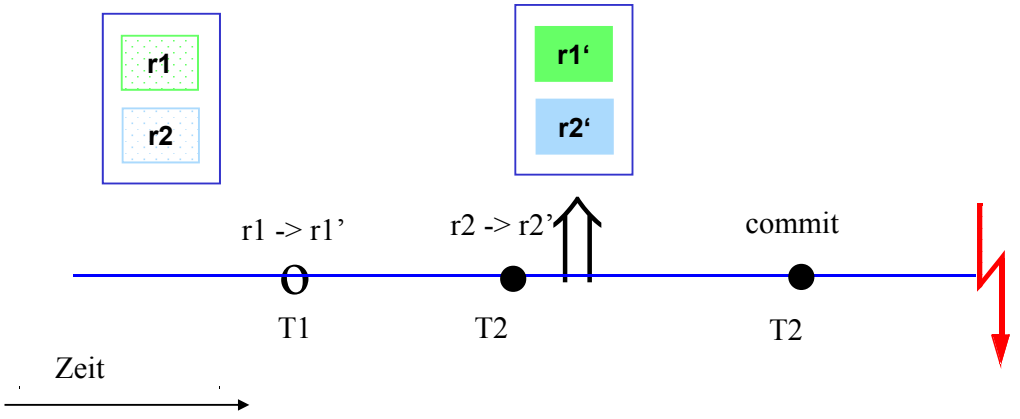
Abhängigkeiten zwischen Systemkomponenten



- Logging/Recovery <-> Sperrverwaltung
  - Log-Granulat muss i.a. kleiner oder gleich dem Sperrgranulat sein!
- Logging/Recovery <-> Einbringstrategie für Änderungen
  - direkt (NonAtomic, Update-in-Place)
  - indirekt (Atomic), Bsp.: Schattenspeicherkonzept
- Logging/Recovery <-> Systempufferverwaltung
  - Verdrängen 'schmutziger' Seiten (Steal vs. NoSteal)
  - Ausschreibstrategie für geänderte Seiten (Force vs. NoForce)

Abhängigkeiten zur Sperrverwaltung

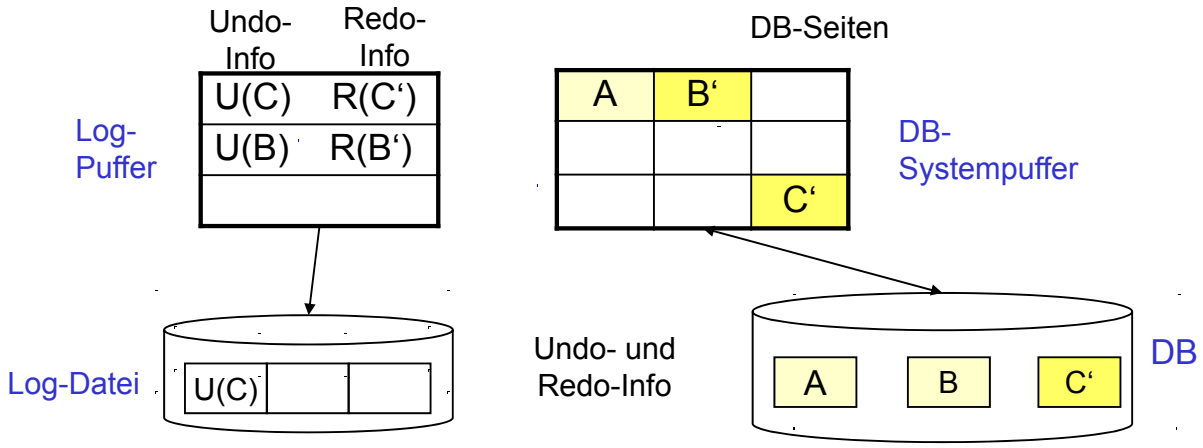
- Log-Granulat muss i.a. kleiner oder gleich dem Sperrgranulat sein
- **Beispiel**: Sperren auf Satzebene, Before- bzw. After-Images auf Seitenebene



=> Undo (Redo) einer Änderung kann parallel durchgeführte Änderungen derselben Seite überschreiben (*lost update*)

Direkte Einbringstrategien: Update in Place

- geänderte Seite wird immer in denselben Block auf Platte zurückgeschrieben
- 'atomares' Zurückschreiben mehrerer geänderter Seiten ist nicht möglich (NonAtomic) -> kein logisches Logging anwendbar

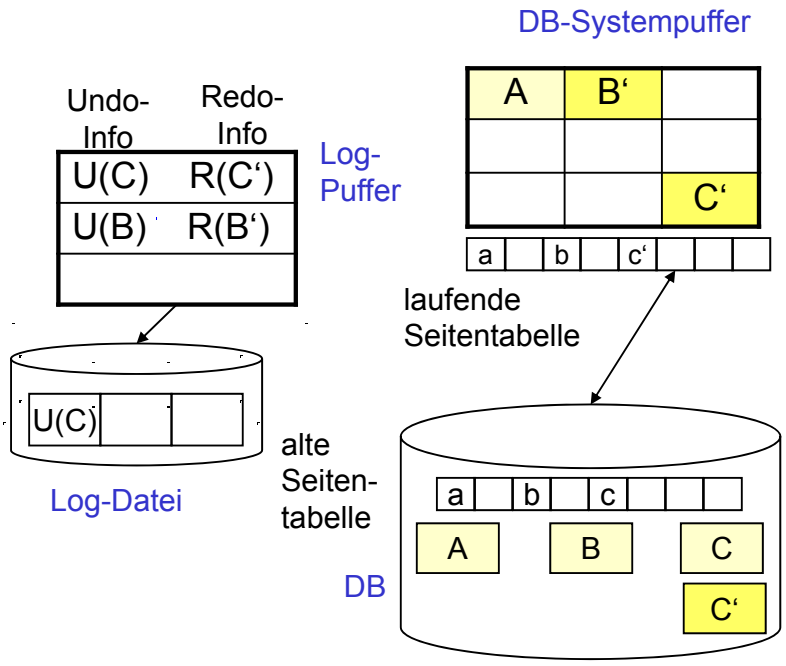


Es sind 2 Prinzipien einzuhalten:

1. *Undo-Regel*: schreibe Undo-Info vor Zurückschreiben unsicherer Änderungen (WAL-Prinzip)
2. *Redo-Regel*: Ausschreiben der Redo-Info spätestens zu COMMIT

Indirekte Einbringstrategien (Atomic)

- Schattenspeicherkonzept
  - geänderte Seite wird in separaten Block auf Platte geschrieben
  - Seitentabelle gibt aktuelle Adresse einer Seite an
  - atomares Einbringen mehrerer Änderungen durch Umschalten von Seitentabellen möglich
  - aktions- oder transaktions-konsistente DB auf Platte
  - logisches Logging anwendbar



- Schwerwiegende Nachteile:
  - aufwändiges Einbringen
  - Seitentabelle kann für große DB nicht mehr im Hauptspeicher gehalten werden
  - Cluster-Eigenschaften werden zerstört
  - erhöhter Speicherplatzbedarf

Abhängigkeiten zur Pufferverwaltung: Ersetzung ‚schmutziger' Seiten

- *Steal*: geänderte Seiten können jederzeit, insbesondere vor Commit der ändernden Transaktion, im Hauptspeicher-Puffer ersetzt und in die permanente DB eingebracht werden
  - \+ große Flexibilität zur Seitenersetzung
  - \- Undo-Recovery vorzusehen (Transaktions-Abbruch, Systemfehler) falls Update-in-Place
- Steal erfordert Einhaltung des ***Write-Ahead-Log (WAL)-Prinzips***:
  vor dem Einbringen einer schmutzigen Änderung in die permanente DB müssen zugehörige Undo-Informationen (z.B. Before-Images) dauerhaft in die Log-Datei geschrieben werden *(Undo-Regel)*
- *NoSteal*
  - \+ keine Undo-Recovery auf der permanenten DB vorzusehen
  - \- Probleme bei langen Änderungstransaktionen (Seiten mit schmutzigen Änderungen dürfen nicht ersetzt werden)

Abhängigkeiten zur Pufferverwaltung: Commit-Behandlung

- *Force:* alle geänderten Seiten werden spätestens beim Commit (vor dem Commit) in die permanente DB durchgeschrieben
  - \+ keine Redo-Recovery nach Rechnerausfall
  - \- hoher Schreibaufwand
  - \- große Systempuffer werden schlecht genutzt
  - \- Antwortzeitverlängerung für Änderungstransaktionen
- *NoForce*:
  - \+ kein Durchschreiben der Änderungen bei Commit: wesentlich leistungsstärker
  - \+ beim Commit werden lediglich Redo-Informationen in die Log-Datei geschrieben
  - \- Redo-Recovery nach Rechnerausfall
- *Commit-Regel (Redo-Regel)*: bevor das Commit einer Transaktion ausgeführt werden kann, sind für ihre Änderungen ausreichende Redo-Informationen (z.B. After-Images) zu sichern

Crash-Recovery-Auswirkungen von Ausschreibstrategien

	Steal	NoSteal
Force		
NoForce		

Force & Nosteal sind inkompatibel

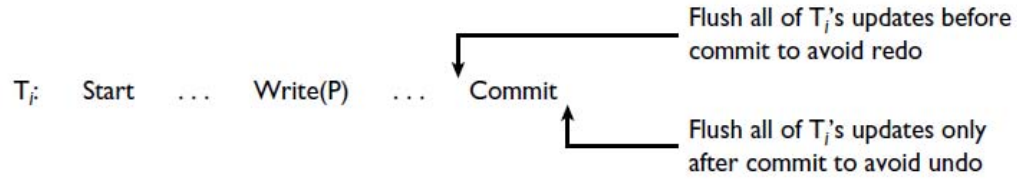


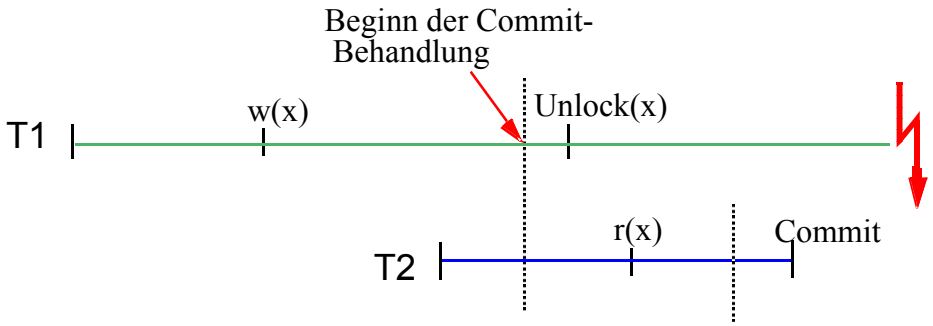
FIGURE 7.11

Avoiding Undo or Redo. Depending on when a transaction's updates are flushed, undo or redo can be avoided.

source: Philip A. Bernstein, Eric Newcomer: *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, 2nd ed. 2009

Commit-Behandlung

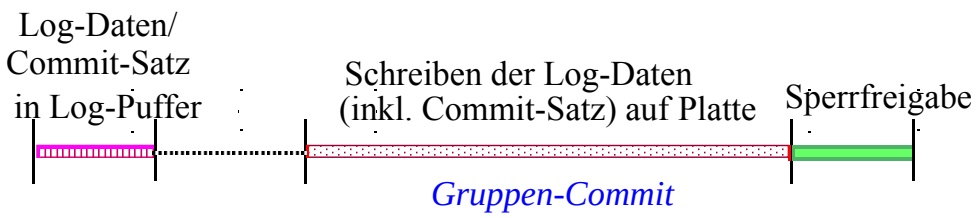
- Änderungen einer Transaktion sind vor Commit zu sichern
  - andere Transaktionen dürfen Änderungen erst sehen, wenn Durchkommen der ändernden Transaktion gewährleistet ist (Problem der 'Cascading Aborts' zu vermeiden)



- Zweiphasige Commit-Bearbeitung
  - ***Phase 1***: Wiederholbarkeit der Transaktion sichern:
    Änderungen ggf. noch sichern und Commit-Satz auf Log schreiben
  - ***Phase 2:*** Änderungen sichtbarmachen (Freigabe der Sperren)
  - Benutzer kann nach Phase 1 vom erfolgreichen Ende der Transaktion informiert werden (Ausgabenachricht)

Gruppen-Commit

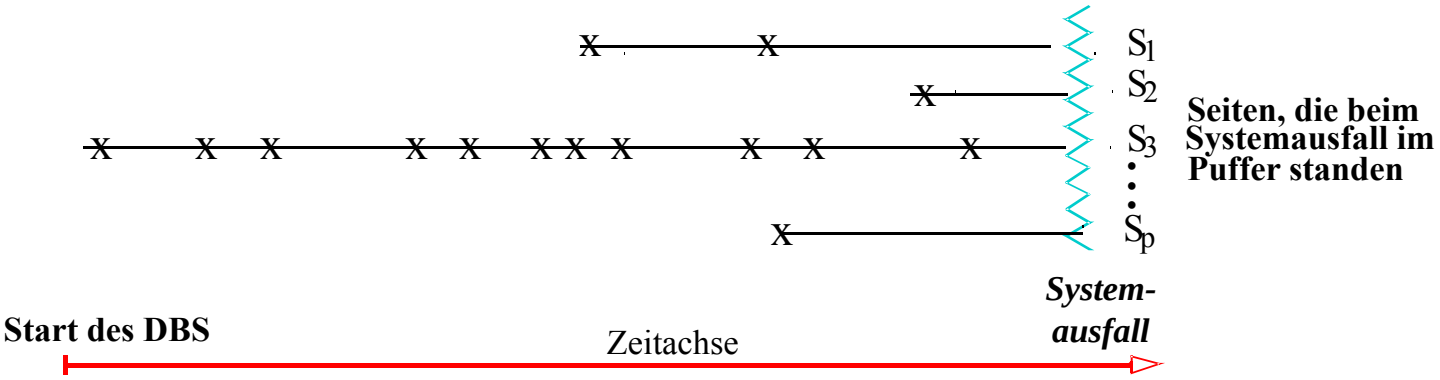
- Log-Datei potentieller Leistungsengpass
  - pro Änderungstransaktion wenigstens 1 Log-E/A
  - max. ca. 100 sequentielle Schreibvorgänge pro Sekunde (1 Platte)
- Gruppen-Commit: gemeinsames Schreiben der Log-Daten von mehreren Transaktionen
  - Pufferung der Log-Daten in Log-Puffer (1 oder mehrere Seiten)
  - Voraussetzung: Eintrags-Logging
  - Ausschreiben des Log-Puffers erfolgt, wenn er voll ist bzw. Timer abläuft
  - i.a. nur geringe Commit-Verzögerung



- erlaubt Reduktion auf 0.1 - 0.2 Log-Seiten pro Transaktion
  => Starke Durchsatzverbesserung

Sicherungspunkte (Checkpoints)

- Sicherungspunkt: Maßnahme zur Begrenzung des Redo-Aufwandes nach Systemfehlern + Begrenzung Log-Umfangs
  - v.a. für NoForce erforderlich
  - ohne Sicherungspunkte sind potentiell alle Änderungen seit Start des DBS zu wiederholen
  - besonders kritisch: häufig geänderte Hot-Spot-Seiten



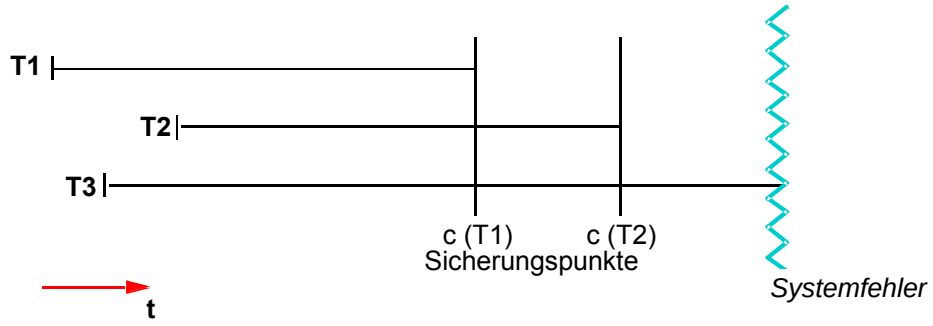
- Log-Adresse des letzten vollständig ausgeführten Checkpoints wird für Recovery in spezieller Restart-Datei geführt

Arten von Sicherungspunkten

- direkte Sicherungspunkte
  - alle geänderten Seiten im DB-Puffer werden auf die permanente DB ausgeschrieben
  - Redo-Recovery beginnt bei letztem Checkpoint
  - Nachteil: lange 'Totzeit' des Systems, da während des Sicherungspunktes keine Änderungen durchgeführt werden können
  - Problem wird durch große Hauptspeicher verstärkt
  - Transaktionskonsistente oder aktionskonsistente Sicherungspunkte
- *Force* kann als spezieller direkter Checkpoint-Typ aufgefasst werden
  (nur Seiten einer Transaktion werden ausgeschrieben => transaktionsorientiert)
- indirekte/unscharfe Sicherungspunkte (Fuzzy Checkpoints)
  - kein Hinauszwingen geänderter Seiten
  - nur Statusinformationen (Pufferbelegung, Menge aktiver Transaktionen, offene Dateien etc.) werden in Log geschrieben
  - sehr geringer Checkpoint-Aufwand
  - i.a. Redo-Informationen vor letztem Sicherungspunkt noch zu berücksichtigen
  - Sonderbehandlung von Hot-Spot-Seiten erforderlich

Transaktionsorientierte Sicherungspunkte

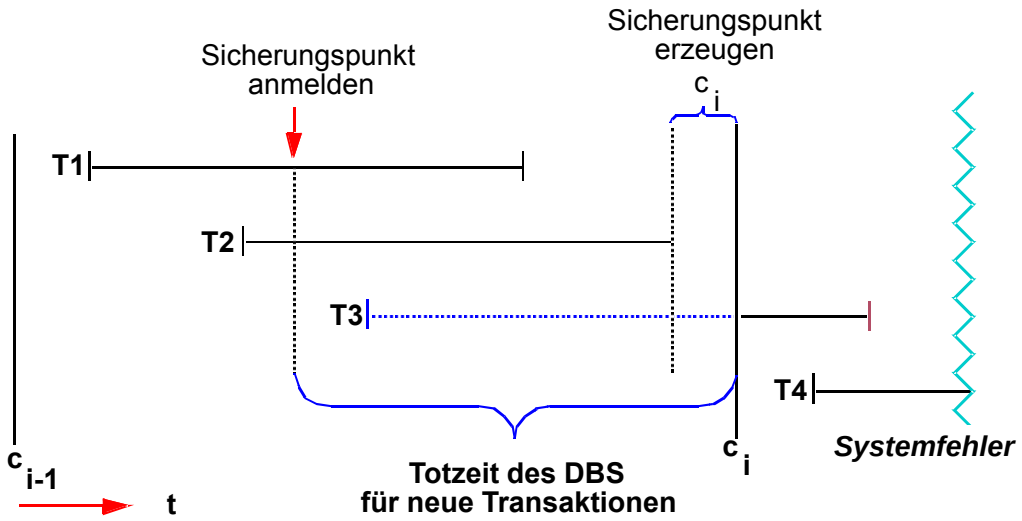
- TOC: Transaction Oriented Checkpoint ≡ Force
- Commit-Behandlung erzwingt Ausschreiben aller geänderten Seiten der Transaktion aus dem Puffer
- Übernahme aller Änderungen in die DB
- Vermerk in Log-Datei



- kein atomares Ausschreiben mehrerer Seiten möglich
  - somit ist bei direkter Seitenzuordnung Undo-Recovery immer vorzusehen (Steal)
  - Abhängigkeit: NonAtomic, Force => Steal

Transaktionskonsistente Sicherungspunkte

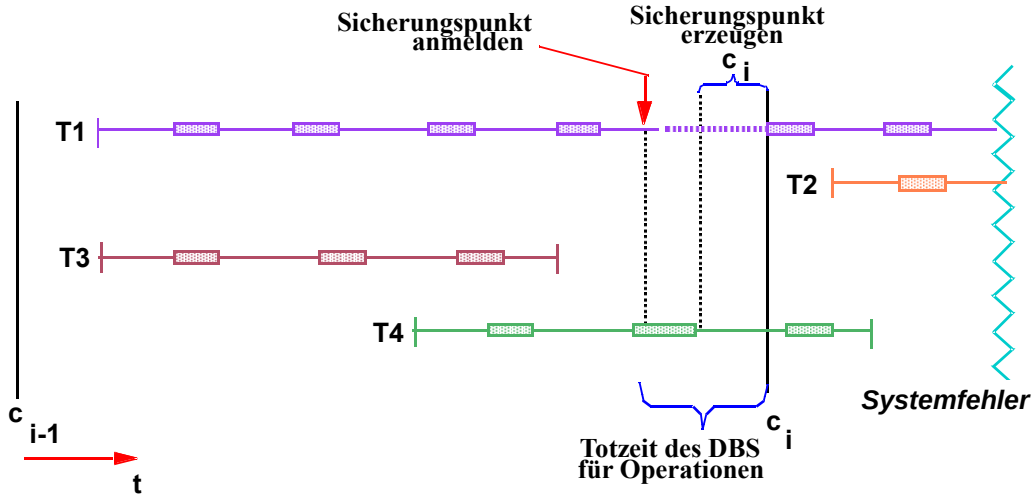
- TCC = Transaction Consistent Checkpoints (logisch konsistent)



- Ausschreiben zu verzögern bis zum Ende aller aktiven Änderungstransaktionen
- neue Änderungstransaktionen müssen warten bis Sicherungspunkt beendet ist
- Crash-Recovery startet bei letztem Sicherungspunkt

Aktionskonsistente Sicherungspunkte

- ACC = Action Consistent Checkpoints

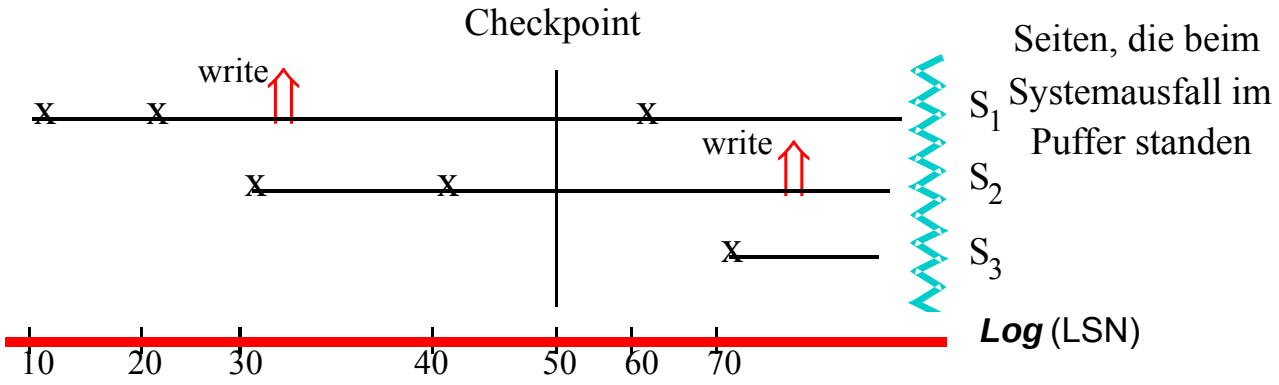


- keine Änderungs-DML-Befehle während Checkpoint
- geringere Totzeiten als bei TCC, dafür Verminderung der Qualität der Sicherungspunkte
- Crash-Recovery wird nicht durch letzten Sicherungspunkt begrenzt
- Abhängigkeit: ACC => Steal

Fuzzy Checkpoints

- DB auf Platte bleibt 'fuzzy', nicht aktionskonsistent
  - nur bei Update-in-Place (NonAtomic) relevant
- sehr schnelle Checkpoint-Durchführung, da kein synchrones Ausschreiben aller geänderten Seiten
  - stoppe Zulassung neuer Update-, Commit und Rollback-Operationen
  - erstelle Liste geänderter Seiten im Puffer
  - erstelle Liste [aktive Transaktion, Pointer zum letzten Log-Satz]
  - schreibe Checkpoint-Satz in den Log mit den Listen
  - fahre mit normaler Verarbeitung fort
  - initiiere asynchrones Ausschreiben der geänderten Seiten
- kein neuer Checkpoint bis alle geänderten Seiten geschrieben
- Checkpoint-Informationen:
  - Liste zum Checkpoint-Zeitpunkt laufender Transaktionen
  - Ids der Seiten, die geändert im Puffer stehen
  - *MinDirtyPageLSN* dieser Seiten (Pufferverwalter vermerkt sich zu jeder geänderten Seite Log-Adresse, DirtyPageLSN, der ersten Änderung seit Einlesen von permanenter DB)

Fuzzy Checkpoints (2)



- Redo-Recovery nach Rechnerausfall beginnt bei *MinDirtyPageLSN* des zuletzt durchgeführten Checkpoints
  - bzw. bei vorletztem Checkpoint
- geänderte Seiten werden asynchron ausgeschrieben
  - nach Ausschreiben einer Seite ggf. MinDirtyPageLSN anpassen

Direkte vs fuzzy Sicherungspunkte

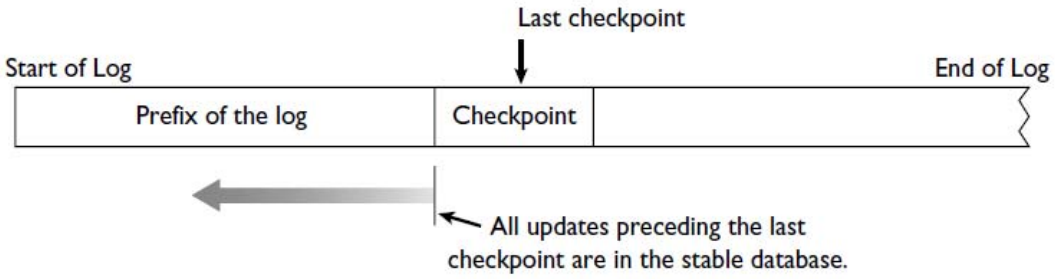


FIGURE 7.18

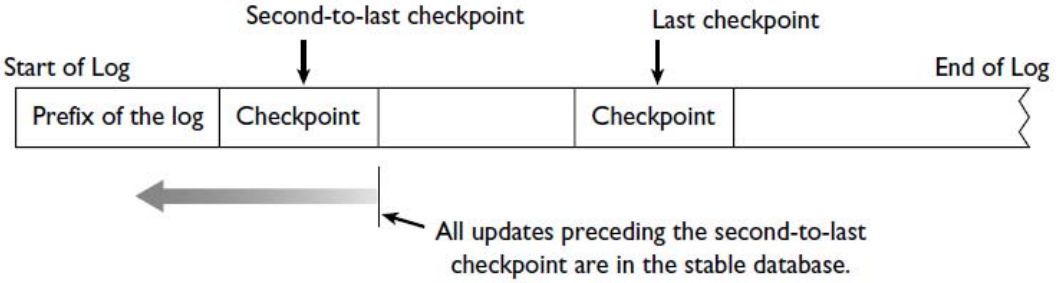
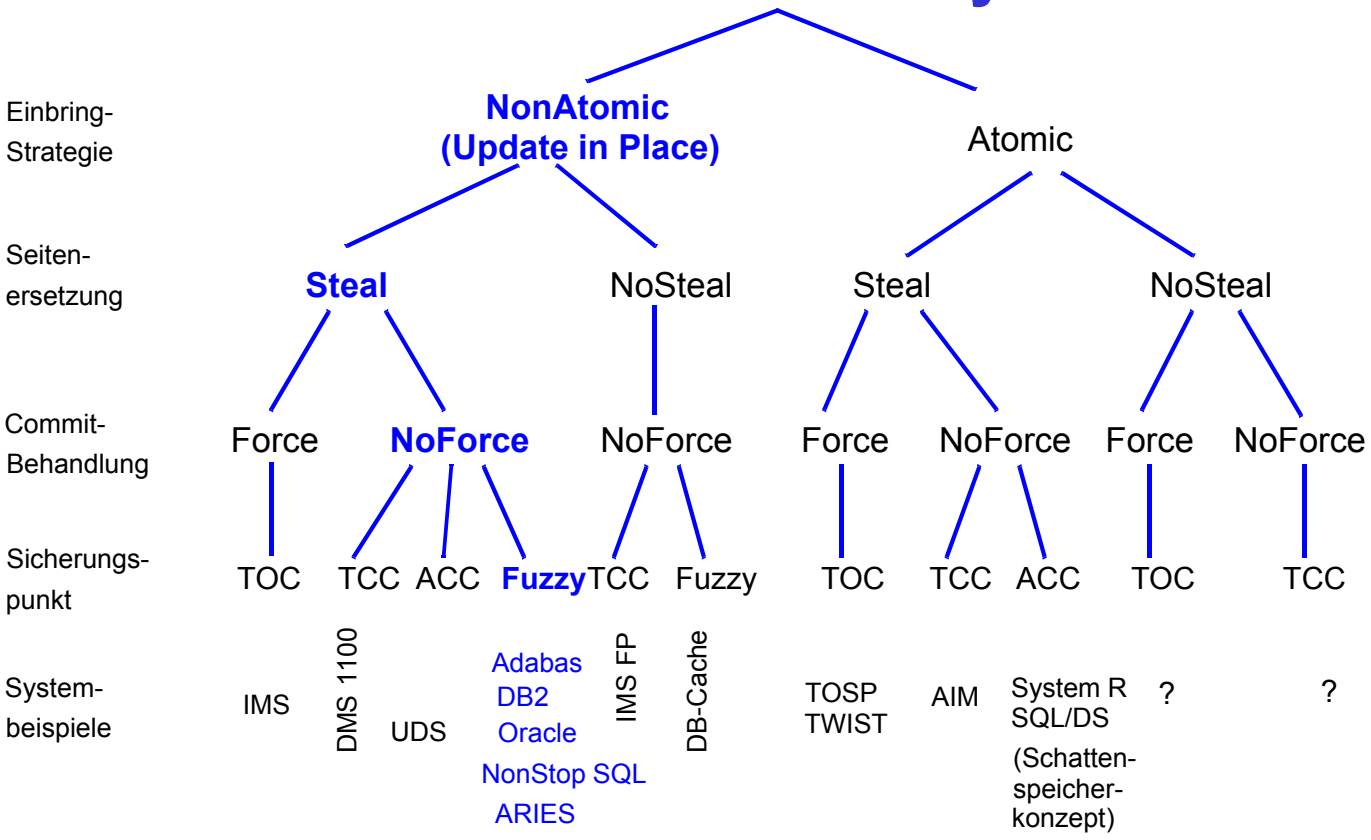


FIGURE 7.19

Fuzzy Checkpointing. After a checkpoint record is written, all dirty cache pages are flushed. The flushes must be completed before the next checkpoint record is written.

source: Philip A. Bernstein, Eric Newcomer: *Principles of Transaction Processing for Systems Professionals.* Morgan Kaufmann 1996

Klassifikation von DB-Recovery-Verfahren



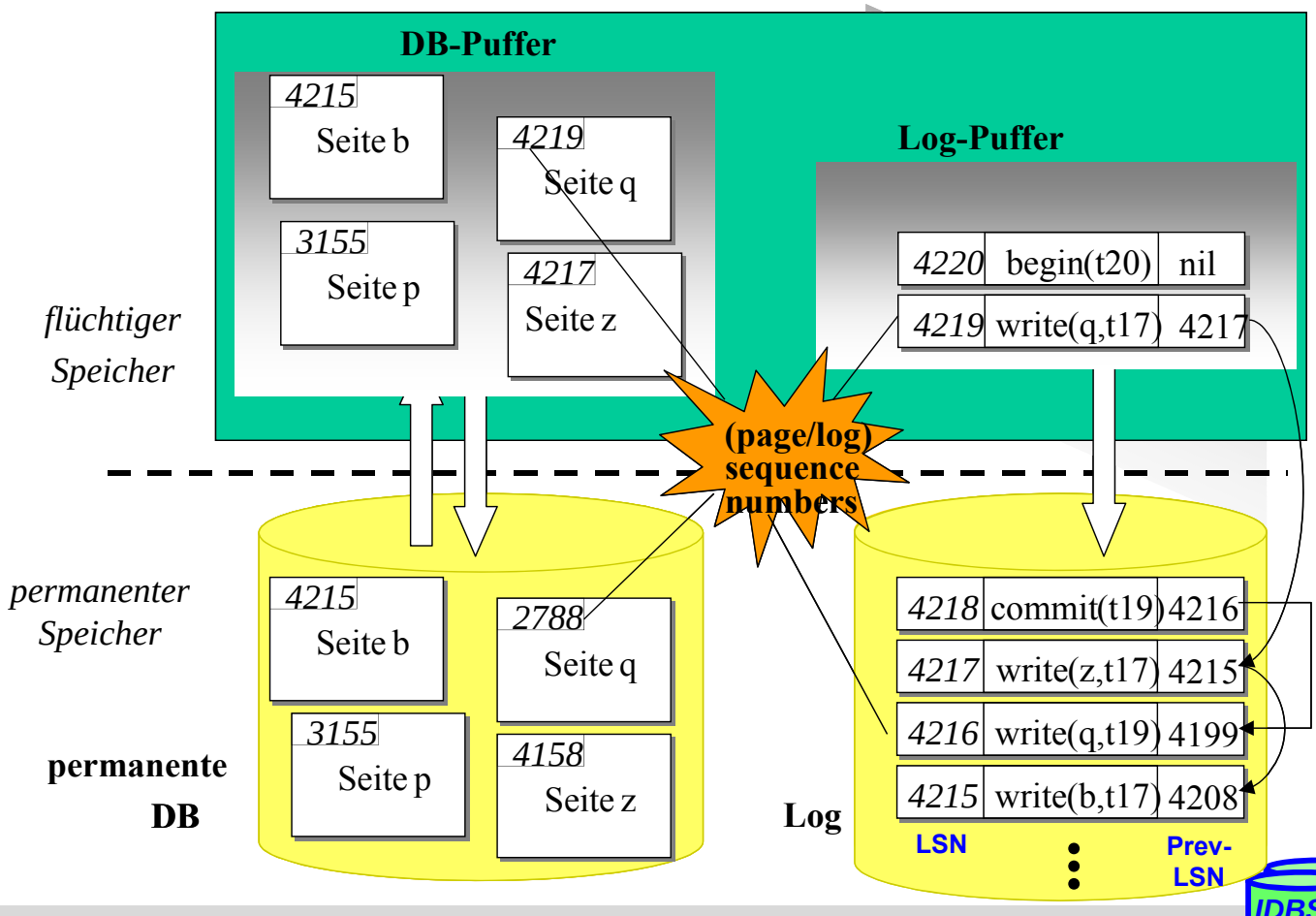
Aufbau der Log-Datei

- i.a. sequenzielle Datei
  - Schreiben neuer Protokolldaten an das aktuelle Dateiende
  - ggf. doppelte Speicherung (Duplex-Logging)
- übliche Satzarten:
  - Begin-of-Transaction (BOT), Commit-Satz, Rollback-Satz
  - Undo-Informationen (z.B. 'Before Images')
  - Redo-Informationen (z.B. 'After Images')
- Checkpoint-Sätze (abhängig von Art der Sicherungspunkte)
  - BEGIN_CHKPT-Satz
  - Checkpoint-Informationen
  - END_CHKPT-Satz
- Log-Sätze einer Transaktion werden rückwärts verkettet (für Transaktions-Undo)
- periodisches Überschreiben alter Log-Daten (Ring-Organisation)

Log-Datei: LSN-Adressierung

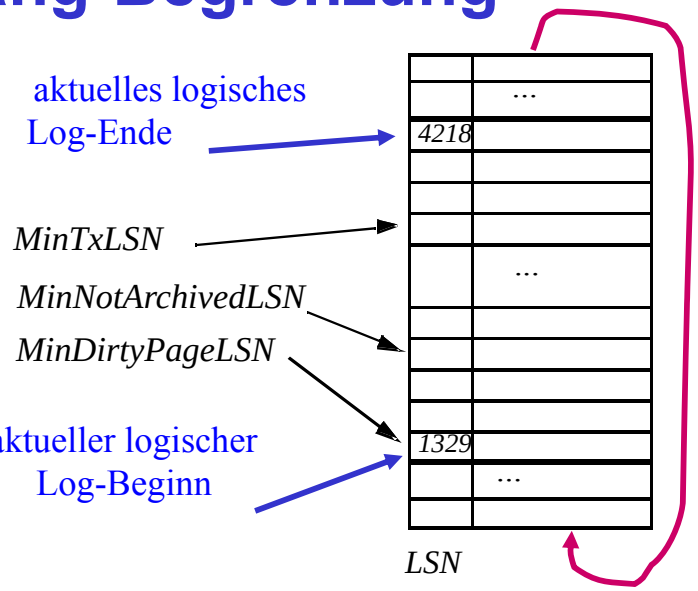
- jeder Log-Satz hat eindeutige Adresse: *LSN (Log Sequence Number)*
  - monoton wachsend !
- jede DB-Seite hat im Seitenkopf ein Feld *PageLSN* mit der LSN derjenigen Änderung, die zuletzt durchgeführt wurde
  - entspricht monoton wachsender Versionsnummer
  - erlaubt Entscheidung darüber, ob ein Log-Satz bei der Recovery anzuwenden ist
- Pufferverwalter merkt sich für geänderte Seiten LSN des Log-Satzes zur ersten Änderung seit Einlesen der Seite (*DirtyPageLSN*)

Beispiel: (Page/Log) Sequence Numbers



Log-Datei: Umfang-Begrenzung

- Log-Daten sind für Crash-Recovery nur begrenzte Zeit relevant



1. Undo-Sätze erfolgreich beendeter Transaktionen werden nicht mehr benötigt
   - pro Transaktion wird LSN ihres BOT-Satzes vermerkt
   - Minimum dieser LSN-Werte laufender Transaktionen (*MinTxLSN*) begrenzt benötigte Undo-Information
2. nach Ausschreiben der Seite in permanente DB wird Redo-Information nicht mehr benötigt
   - Zurücksetzen von DirtyPageLSN und ggf. Anpassen von MinDirtyPageLSN
   - *MinDirtyPageLSN* begrenzt benötigte Redo-Information bzgl Fuzzy Checkpoint
3. Protokollsätze für Platten-Recovery werden auf Archiv-Log gehalten
   - LSN der ältesten noch nicht auf Archiv-Log übertragenen Redo-Information: *MinNotArchivedLSN*

Zusammenfassung

- Fehlerarten: Transaktions-, System- und Gerätefehler
- breites Spektrum von Logging- und Recovery-Verfahren
- Eintrags-Logging ist Seiten-Logging überlegen
  - geringerer Platzbedarf, weniger E/As, Gruppen-Commit
- Update-in-Place-Verfahren sind Atomic-Strategien vorzuziehen
  - erfordern physisches bzw. physiologisches Logging
  - Log-Granulat kleiner oder gleich Sperrgranulat
- NoForce-Strategien sind Force-Verfahren vorzuziehen
  - erfordern Checkpoint-Maßnahmen zur Begrenzung des Redo-Aufwandes
  - 'Fuzzy Checkpoints' erzeugen den geringsten Overhead im Normalbetrieb
- Steal-Methoden sind flexibler als NoSteal
  - verlangen die Einhaltung des WAL-Prinzips
  - erfordern Undo-Aktionen nach einem Rechnerausfall
- sequenzielle Log-Datei
  - Log-Umfang kann begrenzt werden