Logging und Recovery

Norbert Ritter
Datenbanken und Informationssysteme
vsis-www.informatik.uni-hamburg.de

Anforderungen / Begriffe

- Voraussetzungen für die Wiederherstellung der Daten
  - quasi-stabiler Speicher
  - fehlerfreier DBVS-Code
  - fehlerfreie Log-Daten
  - Unabhängigkeit der Fehler
- Recovery-Arten
  1. ***Transaktions-Recovery***
     - Zurücksetzen einzelner (noch nicht abgeschlossener) Transaktionen im laufenden Betrieb (Transaktionsfehler, Deadlock)
     - Arten
       - Vollständiges Zurücksetzen auf Transaktionsbeginn (TA-UNDO)
       - Partielles Zurücksetzen auf Rücksetzpunkt (Savepoint) innerhalb der Transaktion

Anforderungen / Begriffe

- Recovery-Arten (Forts.)
  2. ***Crash-Recovery*** nach Systemfehler
     - Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustands
     - Notwendige Aktionen
       - (partielles) REDO für erfolgreiche Transaktionen (Wiederholung verlorengegangener Änderungen)
       - UNDO aller durch Ausfall unterbrochenen Transaktionen (Entfernen der Änderungen aus der permanenten DB)
  3. ***Medien-Recovery*** nach Gerätefehler
     - Spiegelplatten bzw.
     - Vollständiges Wiederholen (REDO) aller Änderungen (erfolgreich abgeschlossener Transaktionen) auf einer Archivkopie

Anforderungen / Begriffe

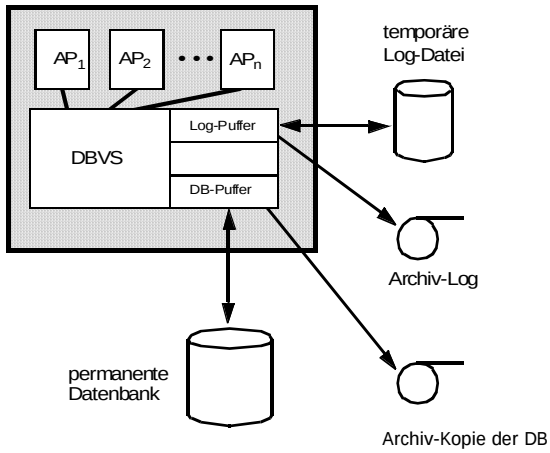
- Recovery-Arten (Forts.)
  4. ***Katastrophen-Recovery***
     - Nutzung einer aktuellen DB-Kopie in einem ‚entfernten' System oder
     - Stark verzögerte Fortsetzung der DB-Verarbeitung mit repariertem/neuem System auf der Basis gesicherter Archivkopien (Datenverlust)

Anforderungen / Begriffe

- Nicht systematisiert
  - R5-Recovery
    - Log-Daten sind fehlerhaft oder DB-Strukturen (ohne Log-Daten) sind unbrauchbar
    - kein TA-konsistenter, bestenfalls aktions- oder gerätekonsistenter Zustand erreichbar
  - R6-Recovery
    - Zusammenfassung aller Maßnahmen außerhalb des Systems
      - Kompensations-TA und
      - manuelle Behandlung der Auswirkungen

Anforderungen / Begriffe

- DB-Recovery – Systemkomponenten

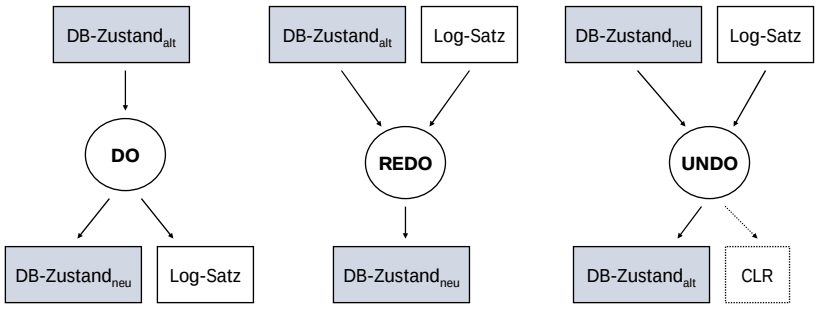


Anforderungen / Begriffe

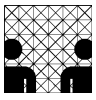
- DB-Recovery – Systemkomponenten (Forts.)
  - Pufferung von Log-Daten im Hauptspeicher (Log-Puffer)
    - Ausschreiben spätestens bei Commit
  - Einsatz der Log-Daten
    - Temporäre Log-Datei zur Behandlung von Transaktions- und Systemfehlern
      - DB + temp. Log ⇒ DB
    - Behandlung von Gerätefehlern
      - Archiv-Kopie + Archiv-Log ⇒ DB

Logging

- Aufgaben
  - Sammlung redundanter Daten bei Änderungen im Normalbetrieb (DO) als Voraussetzung für Recovery
  - Einsatz im Fehlerfall (Undo-, Redo-Recovery)
- Do-Redo-Undo-Prinzip

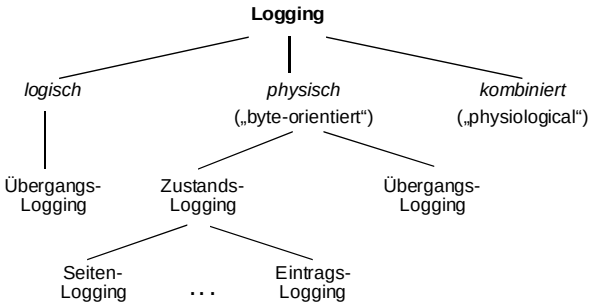


CLR: Compensation Log Record (für Crash während Recovery)



Logging

- Logging kann auf jeder (DBVS-)Systemebene erfolgen
  - Sammeln von ebenenspezifischer Log-Information setzt voraus, dass bei der Recovery die Konsistenzbedingungen der darunter liegenden Abbildungsschicht im DB-Zustand erfüllt sind
- Logging-Verfahren: Klassifikation



Logging

- Logisches Logging
  - Protokollierung der ändernden DML-Befehle mit ihren Parametern
  - Generelles Problem
    - mengenorientierte Aktualisierungsoperation (z. B. DELETE <relation>)
  - UNDO-Probleme v.a. bei nicht-relationalen Systemen
    - z.B. Löschen einer Hierarchie von Set-Ausprägungen – ERASE ALL
  - Voraussetzung
    - nach einem Systemausfall müssen auf der permanenten Datenbank DML-Operationen ausführbar sein, d.h., sie muss wenigstens speicherkonsistent sein (Aktionskonsistenz)
  - verzögerte (indirekte) Einbringstrategie erforderlich

Logging

- Physisches Logging
  - Log-Granulat: Seite vs. Eintrag/Satz
  - Zustands-Logging
    - alte Zustände (Before-Images) und neue Zustände (After-Images) geänderter Objekte werden in die Log-Datei geschrieben
  - Übergangs-Logging
    - Protokollierung der Differenz zwischen Before- und After-Image
  - physisches Logging ist bei direkten und verzögerten Einbringstrategien anwendbar
- Probleme logischer und physischer Logging-Verfahren
  - Logisches Logging: für Update-in-Place nicht anwendbar
  - Physisches, „byte-orientiertes“ Logging: aufwendig und unnötig starr v.a. bezüglich Lösch- und Einfügeoperationen

Logging

- Physiologisches Logging
  - 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 eine Seite
    - Technik ist mit Update-in-Place verträglich

Logging

- Eintrags-Logging vs. Seiten-Logging
  - Vorteile von Eintrags-Logging
    - geringerer Platzbedarf
    - weniger Log-E/As
    - erlaubt bessere Pufferung von Log-Daten (Gruppen-Commit)
    - unterstützt feine Synchronisationsgranulate (Seiten-Logging → Synchronisation auf Seitenebene)
  - Nachteil von Eintrags-Logging
    - Recovery ist komplexer als mit Seiten-Logging
      - z.B. müssen Seiten vor der Anwendung der Log-Sätze wieder in den Speicher eingelesen werden

Logging

- Aufbau der (temporären) Log-Datei
  - Verschiedene Satzarten erforderlich
    - BOT-, Commit-, Abort-Satz
    - Änderungssatz (UNDO-Informationen, z. B. ‚Before-Images', und REDO-Informationen, z. B. ‚After-Images')
    - Sicherungspunkt-Sätze
  - Protokollierung von Änderungsoperationen
    - Struktur der Log-Einträge: [LSN, TAID, PageID, Redo, Undo, PrevLSN]
    - LSN: Log Sequence Number
      - eindeutige Kennung des Log-Eintrags
      - LSNs müssen monoton aufsteigend vergeben werden
      - chronologische Reihenfolge der Protokolleinträge kann dadurch ermittelt werden

Logging

- Aufbau der (temporären) Log-Datei (Forts.)
  - Transaktionskennung TAID der TA, die die Änderung durchgeführt hat
  - PageID
    - Kennung der Seite, auf der die Änderungsoperation vollzogen wurde
    - wenn die Änderung mehr als eine Seite betrifft, müssen entsprechend viele Log-Einträge generiert werden
  - Redo
    - Redo-Information gibt an, wie die Änderung nachvollzogen werden kann
  - Undo
    - Undo-Information beschreibt, wie die Änderung rückgängig gemacht werden kann
  - PrevLSN
    - ist ein Zeiger auf den vorhergehenden Log-Eintrag der jeweiligen TA
    - diesen Eintrag benötigt man aus Effizienzgründen

Logging

- Aufbau der (temporären) Log-Datei (Forts.)
  - Beispiel

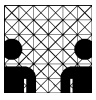
Schritt	T_1	T_2	Log [LSN, TAID, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1, T_1, **BOT**, 0]
2.	r(A, a_1)		
3.		**BOT**	[#2, T_2, **BOT**, 0]
4.		r(C, c_2)	
5.	$a_1 := a_1 - 50$		
6.	w(A, a_1)		[#3, T_1, P_A, A-=50, A+=50, #1]
7.		$c_2 := c_2 + 100$	
8.		w(C, c_2)	[#4, T_2, P_C, C+=100, C-=100, #2]
9.	r(B, b_1)		
10.	$b_1 := b_1 + 50$		
11.	w(B, b_1)		[#5, T_1, P_B, B+=50, B-=50, #3]
12.	**Commit**		[#6, T_1, **Commit**, #5]
13.		r(A, a_2)	
14.		$a_2 := a_2 - 100$	
15.		w(A, a_2)	[#7, T2, P_A, A-=100, A+=100, #4]
16.		**Commit**	[#8, T_2, **Commit**, #7]

Logging

- Aufbau der (temporären) Log-Datei (Forts.)
  - Log ist eine sequentielle Datei
    - Schreiben neuer Protokolldaten an das aktuelle Dateiende
  - Log-Daten sind für Crash-Recovery nur begrenzte Zeit relevant
    - Undo-Sätze für erfolgreich beendete TA werden nicht mehr benötigt
    - nach Einbringen der Seite in die DB wird Redo-Information nicht mehr benötigt
    - Redo-Information für Medien-Recovery ist im Archiv-Log zu sammeln!

Abhängigkeiten der Systemkomponenten

- Überblick über die wichtigsten Abhängigkeiten
  - Einbringstrategie
    - direktes Einbringen von Änderungen ***(non-atomic)***
    - verzögertes Einbringen von Änderungen ***(atomic)***
  - Ersetzungsstrategie
    - Verdrängen 'schmutziger' Seiten ***(steal)***
    - nur Verdrängung von Seiten abgeschlossener TAs ***(nosteal)***
  - Ausschreibstrategie
    - Ausschreiben notwendigerweise bei Commit ***(force)***
    - Ausschreiben möglicherweise nach Commit ***(noforce)***
  - Wahl des Sperrgranulats
  - Commit-Behandlung

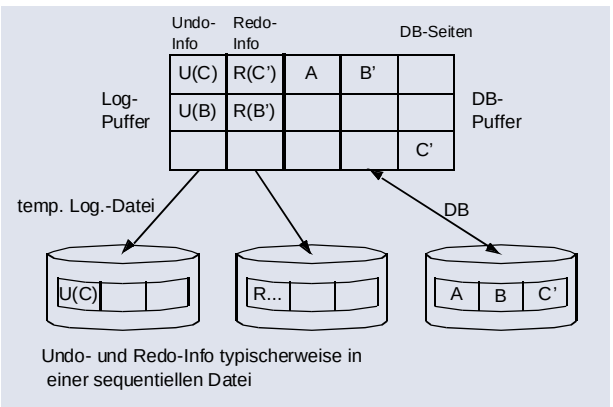


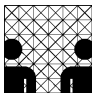
Abhängigkeiten der Systemkomponenten

- Einbringstrategie
  - *non-atomic / direkt / update-in-place*
    - geänderte Seite wird immer in denselben Block auf Platte zurückgeschrieben
    - Schreiben ist gleichzeitig Einbringen in die permanente DB
    - ‚atomares' Einbringen mehrerer geänderter Seiten ist nicht möglich (non-atomic)
    - Forderungen
      - WAL-Prinzip: *Write Ahead Log* für Undo-Info; U(B) vor B' (vgl. nachfolgende Folie)
      - Ausschreiben der Redo-Info spätestens bei Commit; R(C') + R(B') vor Commit (vgl. nachfolgende Folie)

Abhängigkeiten der Systemkomponenten

- Einbringstrategie (Forts.)
  - *non-atomic / direkt / update-in-place (Forts.)*



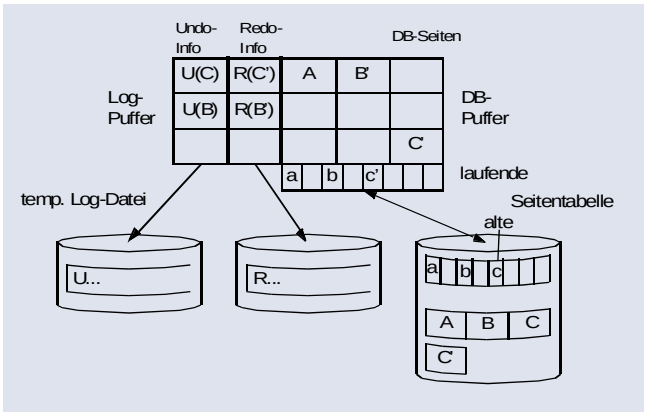


Abhängigkeiten der Systemkomponenten

- Einbringstrategie (Forts.)
  - *atomic / verzögert*
    - z. B. in System R, SQL/DS
    - geänderte Seite wird in separaten Block auf Platte geschrieben, Einbringen in die DB erfolgt später
    - Seitentabelle gibt aktuelle Adresse einer Seite an
    - verzögertes, atomares Einbringen mehrerer Änderungen ist durch Umschalten von Seitentabellen möglich
    - aktions- oder transaktionskonsistente DB auf Platte
    - logisches Logging anwendbar
    - Forderungen
      - WAL-Prinzip bei verzögertem Einbringen:
        U(C) + U(B) vor Sicherungspunkt
      - R(C') + R(B') spätestens bei Commit

Abhängigkeiten der Systemkomponenten

- Einbringstrategie (Forts.)
  - *atomic / verzögert (Forts.)*



Abhängigkeiten der Systemkomponenten

- Ersetzungsstrategie
  - Problem: Ersetzung ‚schmutziger' Seiten
  - *steal*
    - geänderte Seiten können jederzeit, insbesondere vor EOT der ändernden TA, ersetzt und in die permanente DB eingebracht werden
    - große Flexibilität zur Seitenersetzung
    - Undo-Recovery vorzusehen (TA-Abbruch, Systemfehler)
    - *steal* erfordert Einhaltung des WAL-Prinzips
    - vor dem Einbringen einer schmutzigen Änderung müssen zugehörige Undo-Informationen (z. B. Before-Images) in die Log-Datei geschrieben werden
  - *nosteal*
    - Seiten mit schmutzigen Änderungen dürfen nicht ersetzt werden
    - es ist keine Undo-Recovery auf der permanenten DB vorzusehen
    - Probleme bei langen Änderungs-TA

Abhängigkeiten der Systemkomponenten

- Ausschreibstrategie
  - *force*
    - alle geänderten Seiten werden spätestens bei EOT (Commit) in die permanente DB eingebracht (Durchschreiben)
    - keine Redo-Recovery nach Rechnerausfall
    - hoher Schreibaufwand
    - große DB-Puffer werden schlecht genutzt
    - Antwortzeitverlängerung für Änderungs-TA
  - *noforce*
    - kein Durchschreiben der Änderungen bei EOT
    - beim Commit werden lediglich Redo-Informationen in die Log-Datei geschrieben
    - Redo-Recovery nach Rechnerausfall
  - Commit-Regel
    - bevor das Commit einer TA ausgeführt werden kann, sind für ihre Änderungen ausreichende Redo-Informationen *(z. B. After-Images)* zu sichern

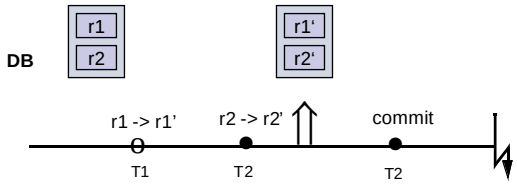
Abhängigkeiten der Systemkomponenten

- Auswirkungen von Ersetzungs- und Ausschreibstrategie auf Recovery-Maßnahmen

	steal	nosteal
force	UNDO NO REDO	NO UNDO NO REDO
noforce	UNDO REDO	NO UNDO REDO

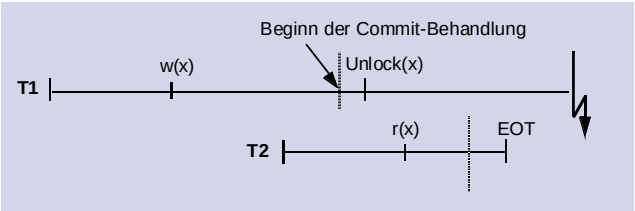
Abhängigkeiten der Systemkomponenten

- Sperrverwaltung
  - Log-Granulat muss kleiner oder gleich 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)*



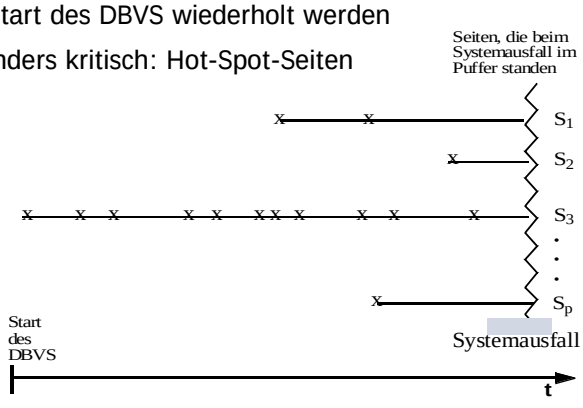
Abhängigkeiten der Systemkomponenten

- Commit-Behandlung
  - Forderungen
    - Änderungen einer TA sind mit Commit zuzusichern
    - andere TA dürfen Änderungen erst sehen, wenn ‚Durchkommen' der ändernden TA gewährleistet ist (Problem des rekursiven Zurücksetzens)



Sicherungspunkte

- Motivation: Sicherungspunkte / Checkpoints
  - Maßnahme zur Begrenzung des Redo-Aufwandes nach Systemfehlern (noforce)
  - ohne Sicherungspunkte müssten potentiell alle Änderungen seit Start des DBVS wiederholt werden
  - Besonders kritisch: Hot-Spot-Seiten



Sicherungspunkte

- Verwaltungsinformation
  - Log-Datei
    - BEGIN_CHKPT-Satz
    - Sicherungspunkt-Informationen, u. a. Liste der aktiven TA
    - END_CHKPT-Satz
  - Log-Adresse des letzten Sicherungspunkt-Satzes wird in spezieller Restart-Datei geführt
- Sicherungspunkte und Einbringverfahren
  - *atomic*
    - Zustand der permanenten DB beim Crash entspricht dem zum Zeitpunkt des letzten erfolgreichen Sicherungspunktes
  - *non-atomic*
    - Zustand der permanenten DB enthält alle ausgeschriebenen (eingebrachten) Änderungen bis zum Crash

Sicherungspunkte

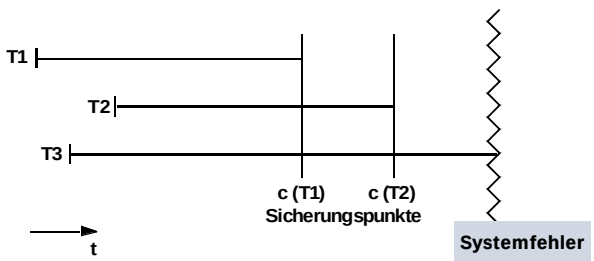
- Arten von Sicherungspunkten
  - Direkte Sicherungspunkte
    - alle geänderten Seiten im DB-Puffer werden in die permanente DB eingebracht
    - Redo-Recovery beginnt bei letztem Sicherungspunkt
    - Nachteil: lange ‚Totzeit' des Systems, da während des Sicherungspunktes keine Änderungen durchgeführt werden können
    - Problem wird durch große Hauptspeicher verstärkt
    - *Transaktionskonsistente* oder *aktionskonsistente* Sicherungspunkte

Sicherungspunkte

- Arten von Sicherungspunkten (Forts.)
  - Indirekte/Unscharfe Sicherungspunkte (Fuzzy Checkpoints)
    - kein Hinauszwingen geänderter Seiten
    - nur Statusinformationen (Pufferbelegung, Menge aktiver TA, offene Dateien etc.) werden in die Log-Datei geschrieben
    - sehr geringer Sicherungspunkt-Aufwand
    - Redo-Informationen vor letztem Sicherungspunkt sind i. allg. noch zu berücksichtigen
    - Sonderbehandlung von Hot-Spot-Seiten

Sicherungspunkte

- Transaktionsorientierte Sicherungspunkte
  - Force kann als spezieller Sicherungspunkt-Typ aufgefasst werden: nur Seiten einer TA werden ausgeschrieben
  - Sicherungspunkt bezieht sich immer auf genau eine TA (TOC = transaction-oriented checkpoint ≡ *force*)

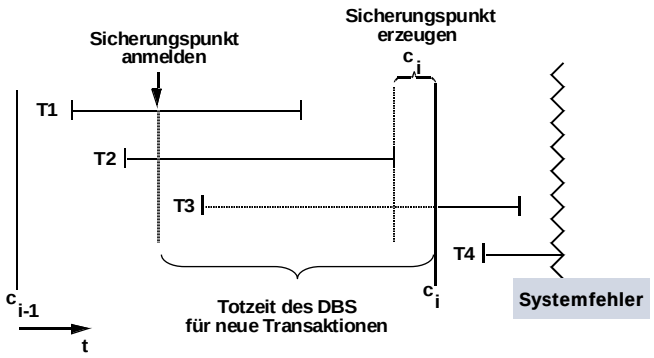


Sicherungspunkte

- Transaktionsorientierte Sicherungspunkte (Forts.)
  - Eigenschaften
    - EOT-Behandlung erzwingt das Ausschreiben aller geänderten Seiten der TA aus dem DB-Puffer
      - Übernahme aller Änderungen in die DB
      - Vermerk in Log-Datei
    - nur *atomic* ermöglicht atomares Einbringen mehrerer Seiten
    - zumindest bei direktem Einbringen der Seiten ist demnach Undo-Recovery vorzusehen *(steal)*
  - Abhängigkeit: *non-atomic*, *force* => *steal*

Sicherungspunkte

- Transaktionskonsistente Sicherungspunkte
  - Sicherungspunkt bezieht sich immer auf alle TA
    (TCC = transaction-consistent checkpoint: logisch konsistent)

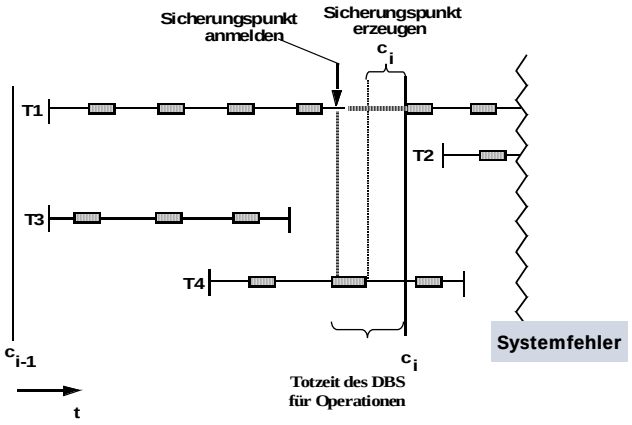


Sicherungspunkte

- Transaktionskonsistente Sicherungspunkte (Forts.)
  - Eigenschaften
    - Ausschreiben ist bis zum Ende aller aktiven Änderungs-TA zu verzögern
    - neue Änderungs-TA müssen warten, bis Erzeugung des Sicherungspunkts beendet ist
    - Crash-Recovery startet bei letztem Sicherungspunkt

Sicherungspunkte

- Aktionskonsistente Sicherungspunkte
  - Sicherungspunkt bezieht sich immer auf alle TA (ACC = action-consistent checkpoint: speicherkonsistent)



Sicherungspunkte

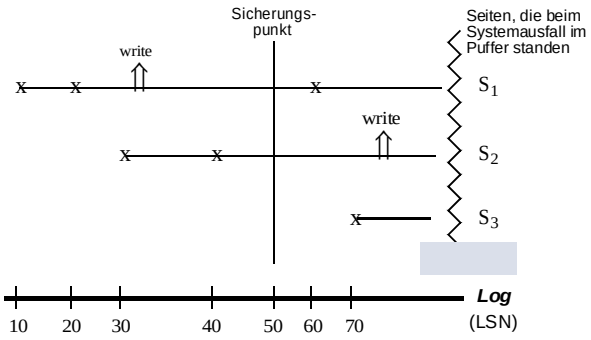
- Aktionskonsistente Sicherungspunkte (Forts.)
  - Eigenschaften
    - keine Änderungsanweisungen während des Sicherungspunktes
    - geringere Totzeiten als bei TCC, dafür Verminderung der Qualität der Sicherungspunkte
    - Crash-Recovery wird nicht durch letzten Sicherungspunkt begrenzt
  - Abhängigkeit: ACC => *steal*

Sicherungspunkte

- Unscharfe Sicherungspunkte (Fuzzy Checkpoints)
  - DB auf Platte bleibt ‚fuzzy', nicht aktionskonsistent
    - nur bei Update-in-Place *(non-atomic)* relevant
  - Problem
    - Bestimmung der Log-Position, an der Redo-Recovery beginnen muss
    - Pufferverwalter vermerkt sich zu jeder geänderten Seite StartLSN, d. h. Log-Satz-Adresse der ersten Änderung seit Einlesen von Platte
    - Redo-Recovery nach Crash beginnt bei MinDirtyPageLSN (= MIN(StartLSN))
  - Sicherungspunkt-Information
    - MinDirtyPageLSN, Liste der aktiven TA und ihrer StartLSNs, ...

Sicherungspunkte

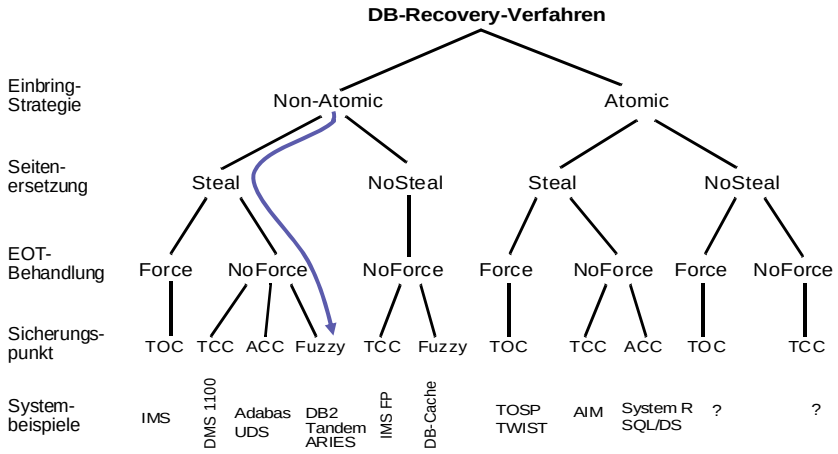
- Unscharfe Sicherungspunkte (Forts)



  - geänderte Seiten werden asynchron ausgeschrieben
    - ggf. Kopie der Seite anlegen (für Hot-Spot-Seiten)
    - Seite ausschreiben
    - StartLSN anpassen / zurücksetzen

Sicherungspunkte

- Kombinierbarkeit der Verfahren



Recovery

- Test zur Fehlerbehandlung

	Datenseite bereits in DB eingebracht	Log-Satz bereits in Log-Datei geschrieben	TA nicht beendet, ggf. Zurücksetzung	TA abgeschlossen, ggf. Wiederholung
1	nein	nein	**a**	**e**
2	nein	ja	**a**	**c**
3	ja	nein	**d**	**e**
4	ja	ja	**b**	**a**

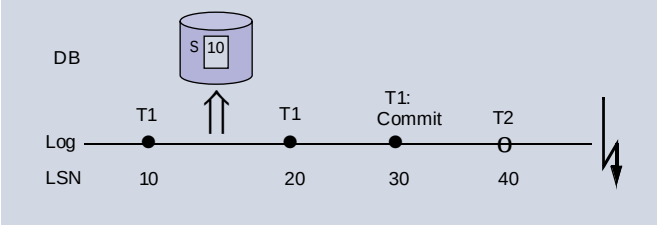
  - Mögliche Antworten:
    a. tu' nichts
    b. benutze die UNDO-Information und setze zurück
    c. benutze die REDO-Information und wiederhole
    d. WAL-Prinzip verhindert diese Situation
    e. 2PC verhindert diese Situation

Recovery

- Nutzung von LSNs
  - Herausforderung
    - beim Restart zu entscheiden, ob für die Seite Recovery-Maßnahmen anzuwenden sind oder nicht (ob der alte oder bereits der geänderte Zustand auf dem Externspeicher vorliegt)
    - dazu wird auf jeder Seite B (im Seitenkopf) die LSN des jüngsten diese Seite betreffenden Log-Eintrags L gespeichert (PageLSN (B) := LSN (L))
  - Entscheidungsprozedur
    - Restart hat eine Redo- und eine Undo-Phase
    - Redo ist nur erforderlich, wenn
      Seiten-LSN $<$ LSN des Redo-Log-Satzes
    - Undo ist nur erforderlich, wenn
      Seiten-LSN $\geq$ LSN des Undo-Log-Satzes

Recovery

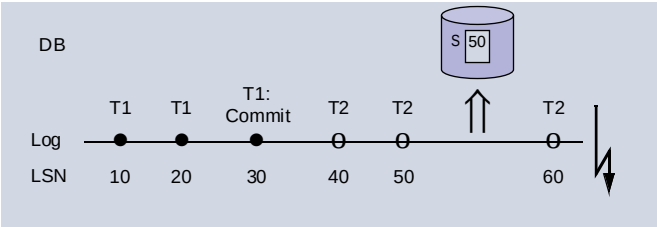
- Nutzung von LSNs (Forts.)
  - Vereinfachte Anwendung: Seitensperren



- Redo von T1: S(10) = T1(10): –
  S(10) < T1(20): Redo, S(20)
- Undo von T2: S(20) < T2(40): –
- Seiten-LSN wird demnach bei Redo aktualisiert (wächst monoton)

Recovery

- Nutzung von LSNs (Forts.)
  - Vereinfachte Anwendung: Seitensperren (Forts.)



- Redo von T1: S(50) > T1(10): –
  S(50) > T1(20): –
- Undo von T2: S(50) < T2(60): –
  S(50) ≥ T2(50): Undo
  S(50) ≥ T2(40): Undo

Recovery

- Nutzung von LSNs (Forts.)
  - Undo erfolgt in LIFO-Reihenfolge
    - ‚Undos' müssen speziell behandelt werden, so dass wiederholte Ausführung zum gleichen Ergebnis führt (Idempotenz)
    - Zustandslogging und LIFO-Reihenfolge gewährleisten Idempotenz

Recovery

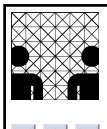
- Crash-Recovery
  - Ziel
    - Herstellung des jüngsten transaktionskonsistenten DB-Zustandes aus permanenter DB und temporärer Log-Datei
  - bei Update-in-Place (*non-atomic*)
    - Zustand der permanenten DB nach Crash unvorhersehbar („chaotisch")
    - daher nur physische Logging-Verfahren anwendbar
    - ein Block der permanenten DB ist entweder
      - aktuell
      - oder veraltet *(noforce)* → Redo
      - oder ‚schmutzig' *(steal)* → Undo

Recovery

- Crash-Recovery (Forts.)
  - bei *atomic*
    - permanente DB entspricht Zustand des letzten erfolgreichen Einbringens (Sicherungspunkt)
    - zumindest aktionskonsistent → DML-Befehle ausführbar (logisches Logging)
    - *force:* kein Redo
    - *noforce:*
      - transaktionskonsistentes Einbringen → Redo, jedoch kein Undo
      - aktionskonsistentes Einbringen → Undo + Redo

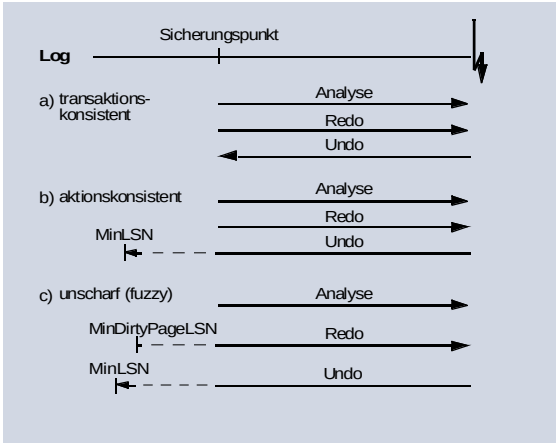
Recovery

- Allgemeine Restart-Prozedur
  - temporäre Log-Datei wird 3-mal gelesen
    1. Analyse-Phase
       - vom letzten Sicherungspunkt bis zum Log-Ende
       - Bestimmung von Gewinner- und Verlierer-TA sowie der Seiten, die von ihnen geändert wurden
    2. Redo-Phase
       - Vorwärtslesen des Log: Startpunkt abhängig vom Sicherungspunkt-Typ
       - selektives Redo bei Seitensperren *(redo winners)* oder vollständiges Redo *(repeating history)* möglich
    3. Undo-Phase
       - Rücksetzen der Verlierer-TA durch Rückwärtslesen des Logs bis zum BOT-Satz der ältesten Verlierer-TA



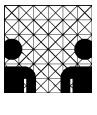
Recovery

- Allgemeine Restart-Prozedur (Forts.)



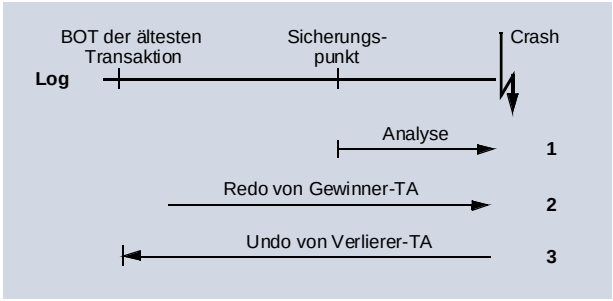
Recovery

- Restart-Prozedur bei Update-in-Place
  - Attribute: non-atomic, steal, noforce, fuzzy checkpoints
  - Ablauf
    1. Analyse-Phase
       - vom letzten Sicherungspunkt bis zum Log-Ende
    2. Redo-Phase
       - Startpunkt abhängig vom Sicherungspunkt-Typ: hier MinDirtyPageLSN
       - selektives Redo: nur Wiederholung der Änderungen der Gewinner-TA
    3. Undo-Phase
       - Rücksetzen der Verlierer-TA bis MinLSN

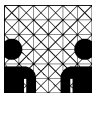


Recovery

- Restart-Prozedur bei Update-in-Place (Forts.)



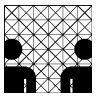
  - Aufwandsaspekte
    - für Schritt 2 und 3 sind betroffene DB-Seiten einzulesen
    - LSN der Seiten zeigen, ob Log-Informationen anzuwenden sind
    - am Ende sind alle geänderten Seiten wieder auszuschreiben, bzw. es wird ein Sicherungspunkt erzeugt



Recovery

- Redo-Recovery
  - Notwendigkeit
    - bei physischem und physiologischem Logging
      - Redo-Aktion für Log-Satz L wird über PageLSN der betroffenen Seite B angezeigt

        if (B nicht gepuffert) then (lies B in den Hauptspeicher ein);
        if LSN (L) > PageLSN (B) then do;
        Redo (Änderung aus L);
        PageLSN (B) := LSN (L);
        end;
    - wiederholte Anwendung des Log-Satzes (z.B. nach mehrfachen Fehlern) erhält Korrektheit (Redo-Idempotenz)

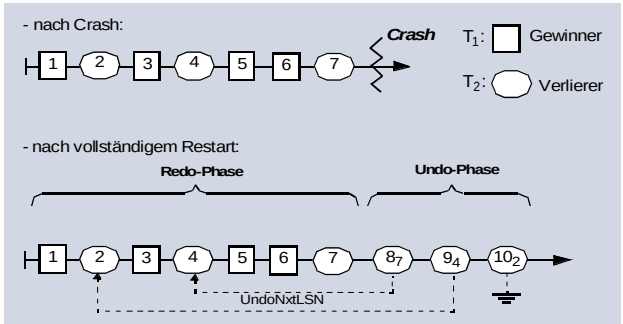


Recovery

- Fehlertoleranz des Restart (Forts.)
  - Forderung: Idempotenz des Restart (Forts.)
    - Gewährleistung der Idempotenz der Undo-Phase erfordert ein neues Konzept: CLR = Compensation Log Record
      - Änderungen der DB sind durch Log-Einträge abzusichern - und zwar im Normalbetrieb und beim Restart!
  - Optimierte Lösung
    - Einsatz von CLRs bei allen Undo-Operationen: Rollback und Undo-Phase
    - in der Redo-Phase: vollständiges Redo von Gewinnern und Verlierern („repeating history“)

Recovery

- Compensation Log Records (CLR)
  - Optimierte Lösung (Forts.)



Recovery

- Compensation Log Records (Forts.)
  - Optimierte Lösung (Forts.)
    - schematische Darstellung der Log-Datei (zu Abbildung auf vorangegangener Folie)
      - die Redo-Information eines CLR entspricht der während der Undo-Phase ausgeführten Undo-Operation
      - CLR-Sätze werden bei erneutem Restart benötigt (nach Crash beim Restart); ihre Redo-Information wird während der Redo-Phase angewendet; dabei werden Seiten-LSNs geschrieben → die Redo-Phase ist idempotent**!**
      - CLRs benötigen keine Undo-Information, da sie während nachfolgender Undo-Phasen übersprungen werden (UndoNxtLSN)