Transaktionale Informationssysteme

-

5. Logging und Recovery

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

Anforderungen / Begriffe (1)

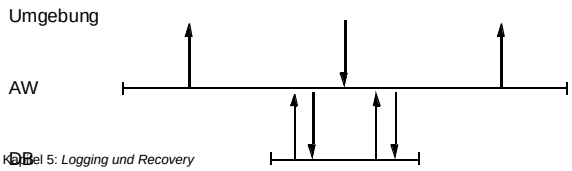
- Aufgabe des DBVS
  - Automatische Behandlung aller erwarteten Fehler
- Erwartete Fehler
  - DB-Operation wird zurückgewiesen
  - Commit wird nicht akzeptiert
  - Stromausfall
  - Geräte funktionieren nicht (Spur, Zylinder, Platte defekt)
  - ...
- Besonderheiten der DBS-Fehlerbehandlung
  - Begrenzung und Behebung der zur Laufzeit möglichen Fehler (wie auch bei anderen fehlertoleranten Systemen)
  - „Reparatur" der statischen Struktur der DB

Anforderungen / Begriffe (2)

- Allgemeine Probleme
  - Fehlererkennung
  - Fehlereingrenzung
  - Abschätzung des Schadens
  - Durchführung der Recovery
- Fehlermodell von zentralisierten DBVS
  - Transaktionsfehler
  - Systemfehler
  - Gerätefehler
  - Katastrophen
- Voraussetzung
  - Sammeln redundanter Informationen während des Normalbetriebs

Anforderungen / Begriffe (3)

- Transaktionsparadigma verlangt
  - Alles-oder-Nichts-Eigenschaft von Transaktionen
  - Dauerhaftigkeit erfolgreicher Änderungen
- Zielzustand nach erfolgreicher Recovery
  - jüngster transaktionskonsistenter DB-Zustand
    - 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
- Zustand der Systemumgebung?
  - Betriebssystem, Anwendungssystem, andere Komponenten



Anforderungen / Begriffe (4)

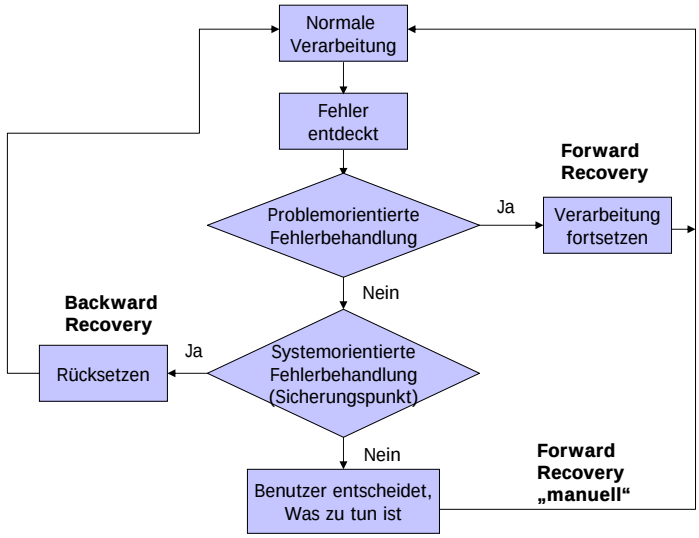
- Wie soll Recovery durchgeführt werden?
  - Forward-Recovery
    - Non-Stop-Paradigma (Prozesspaare usw.)
    - i. allg. nicht anwendbar
    - Fehlerursache häufig falsche Programme, Eingabefehler u. ä.
    - durch Fehler unterbrochene TA sind zurückzusetzen
  - Backward-Recovery
    - setzt voraus, dass auf allen Abstraktionsebenen genau definiert ist, auf welchen Zustand die DB im Fehlerfall zurückzusetzen ist
    - Zurücksetzen auf konsistenten Zustand und Wiederholung

Anforderungen / Begriffe (5)

- Aufwand
  - "A recoverable action is 30% harder and requires 20% more code than a non-recoverable action" (J. Gray)
  - Anweisungs- und TA-Atomarität gefordert
    - zwei Prinzipien der Anweisungs-Atomarität möglich
      - „Do things twice" (vorbereitende Durchführung der Operation; wenn alles OK, erneuter Zugriff und Änderung)
      - „Do things once" (sofortiges Durchführen der Änderung; wenn Fehler auftritt, internes Zurücksetzen)
    - zweites Prinzip wird häufiger genutzt (ist optimistischer und effizienter)

Anforderungen / Begriffe (6)

- Grundsätzliche Vorgehensweise



Anforderungen / Begriffe (7)

- Fehlerarten

Auswirkung eines Fehlers auf	Fehlertyp	Fehler-klassifikation
Eine Transaktion	Verletzung von Systemrestriktionen Verstoß gegen Sicherheitsbestimmungen übermäßige Betriebsmittelanforderungen anwendungsbedingte Fehler z. B. falsche Operationen und Werte	Transaktions-fehler
Mehrere Transaktionen	geplante Systemschließung	
	Schwierigkeiten bei der Betriebsmittelvergabe Überlast des Systems Verklemmung mehrerer Transaktionen	
Alle Transaktionen (das gesamte Systemverhalten)	Systemzusammenbruch mit Verlust der Hauptspeicherinhalte Hardware-Fehler falsche Werte in kritischen Tabellen	Systemfehler
	Zerstörung von Sekundärspeichern	Gerätefehler
	Zerstörung des Rechenzentrums	Katastrophe

Anforderungen / Begriffe (8)

- 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 (9)

- 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 (10)

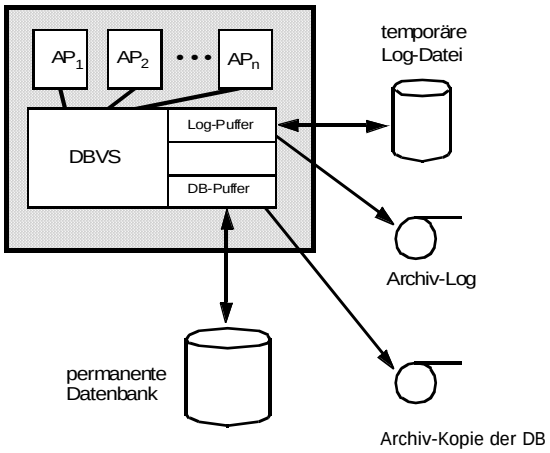
- Recovery-Arten (Forts.)
  4. ***Katastrophen-Recovery***
     - Nutzung einer aktuellen DB-Kopie in einem ‚entfernten' System oder
     - Stark verzögerte Fortsetzung der DB-Verarbeitung mit repariertem/neuem System auf der Basis gesicherter Archivkopien (Datenverlust)

Anforderungen / Begriffe (11)

- 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 (12)

- DB-Recovery – Systemkomponenten

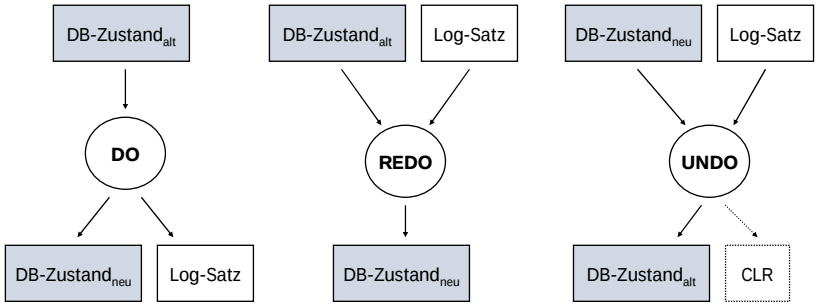


Anforderungen / Begriffe (13)

- 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 $\Rightarrow$ DB
    - Behandlung von Gerätefehlern
      - Archiv-Kopie + Archiv-Log $\Rightarrow$ DB

Logging (1)

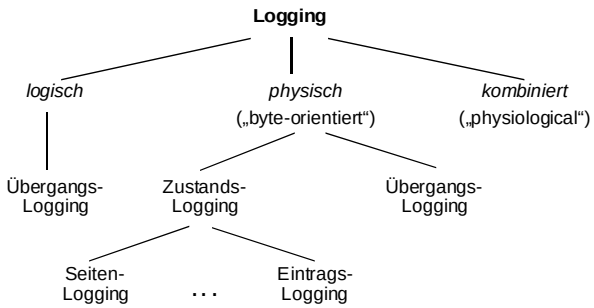
- Aufgaben
  - Sammlung redundanter Daten bei Änderungen im Normalbetrieb (DO) als Voraussetzung für Recovery
  - Einsatz im Fehlerfall (Undo-, Redo-Recovery)
- Do-Redo-Undo-Prinzip



CLR: Compensation Log Record (für Crash während Recovery)

Logging (2)

- 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 (3)

- 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 (4)

- 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 (5)

- 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 (6)

- Anwendungsbeispiel
  - Änderungen bezüglich einer Seite A
    1. Ein Objekt a wird in Seite A eingefügt ($A_1 \rightarrow A_2$)
    2. in A wird ein bestehendes Objekt b_{alt} nach b_{neu} geändert ($A_2 \rightarrow A_3$)

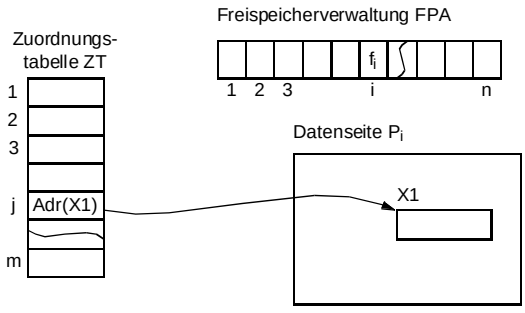
	logisch	*physisch*
Zustände		Protokollierung der Before- und After-Images 1. A_1 und A_2 2. A_2 und A_3
Übergänge	Protokollierung der Operationen mit Parameter 1. Insert (a) 2. Update (b_{alt}, b_{neu})	Differenzen-Logging 1. $A_1 \oplus A_2$ 2. $A_2 \oplus A_3$

Logging (7)

- Anwendungsbeispiel (Forts.)
  - Rekonstruktion von Seiten beim Differenzen-Logging
    - A1 als Anfangs- oder A3 als Endzustand seien verfügbar
    - REDO-Recovery
      - $A_1 \oplus (A_1 \oplus A_2) = A_2$
      - $A_2 \oplus (A_2 \oplus A_3) = A_3$
    - UNDO-Recovery
      - $A_3 \oplus (A_2 \oplus A_3) = A_2$
      - $A_2 \oplus (A_1 \oplus A_2) = A_1$

Logging (8)

- Aufwand bei physischem Zustandslogging
  - einfachste Form der Implementierung: Seiten-Logging



Logging (9)

- Aufwand bei physischem Zustandslogging (Forts.)
  - Operation: STORE X-RECORD (X1)

Aufwand	*Datenseite*	*ZT*	*FPA*	*n Zugriffspfadseiten*
normaler Betrieb (DO)	neues P_i	Adr(X1)	f_i	n DS_{neu}
UNDO-Log	altes P_i	alter Inhalt	alter Inhalt	n DS_{alt}
REDO-Log	neues P_i	Adr(X1)	f_i	n DS_{neu}

Logging (10)

- Bewertung der Logging-Verfahren

	Logging-Aufwand im Normalbetrieb	Restart-Aufwand im Fehlerfall (Crash)
Seitenzustands-Logging	--	+
Seitenübergangs-Logging (Differenzen)	-	+
Eintrags-Logging / physiologisches Logging	+	+
Logisches Logging	++	--

Logging (11)

- 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 (12)

- 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 (13)

- 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 (14)

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

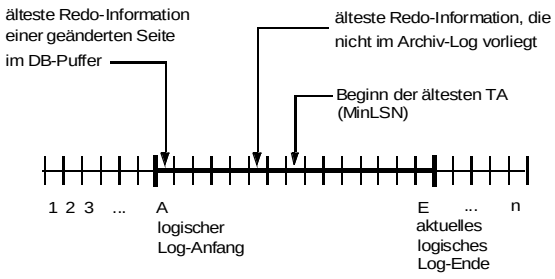
Schritt	T_1	T_2	Log [LSN, TAID, PageID, Redo, Undo, PrevLSN]
1.	**BOT**		[#1, T_1, **BOT**, 0]
2.	r(A, a_1)		
3.		**BOT**	[#2, T_2, **BOT**, 0]
4.		r(C, c_2)	
5.	$a_1 := a_1 - 50$		
6.	w(A, a_1)		[#3, T_1, P_A, A-=50, A+=50, #1]
7.		$c_2 := c_2 + 100$	
8.		w(C, c_2)	[#4, T_2, P_C, C+=100, C-=100, #2]
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 (15)

- 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!

Logging (16)

- Aufbau der (temporären) Log-Datei (Forts.)
  - Ringpuffer-Organisation der Log-Datei



Abhängigkeiten der Systemkomponenten (1)

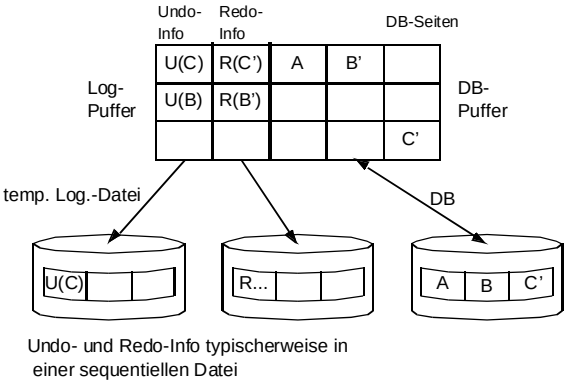
- Überblick über die wichtigsten Abhängigkeiten
  - Einbringstrategie
    - direktes Einbringen von Änderungen ***(non-atomic)***
    - verzögertes Einbringen von Änderungen ***(atomic)***
  - Ersetzungsstrategie
    - Verdrängen 'schmutziger' Seiten ***(steal)***
    - nur Verdrängung von Seiten abgeschlossener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

Abhängigkeiten der Systemkomponenten (2)

- 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 (3)

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



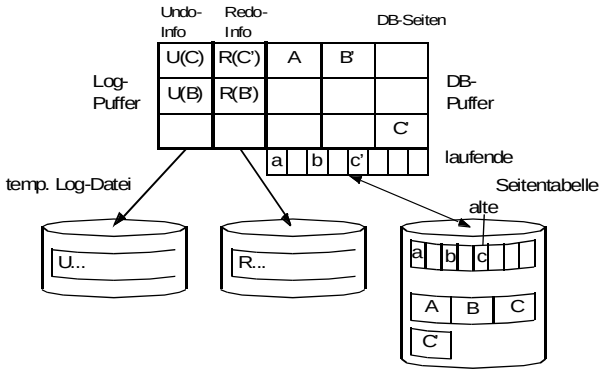
Undo- und Redo-Info typischerweise in einer sequentiellen Datei

Abhängigkeiten der Systemkomponenten (4)

- 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 (5)

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



Abhängigkeiten der Systemkomponenten (6)

- 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 (7)

- 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

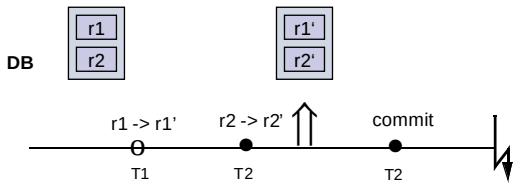
Abhängigkeiten der Systemkomponenten (8)

- Auswirkungen von Ersetzungs- und Ausschreibstrategie auf Recovery-Maßnahmen

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

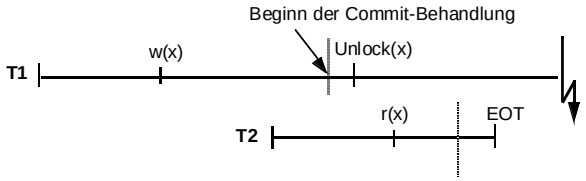
Abhängigkeiten der Systemkomponenten (9)

- 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 (10)

- 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)



Abhängigkeiten der Systemkomponenten (11)

- Commit-Behandlung (Forts.)
  - zweiphasige Commit-Bearbeitung
    - Phase 1: Wiederholbarkeit der TA sichern
      - ggf. noch Änderungen sichern
      - Commit-Satz auf Log schreiben
    - Phase 2: Änderungen sichtbarmachen (Freigabe der Sperren)
    - Benutzer kann nach Phase 1 vom erfolgreichen Ende der TA informiert werden (Ausgabenachricht)
    - Beispiel: Commit-Behandlung bei *force, steal:*
      1. Before-Images auf Log schreiben
      2. Force der geänderten DB-Seiten
      3. After-Images (für Archiv-Log) und Commit-Satz schreiben

      bei NoForce lediglich 3. für erste Commit-Phase notwendig

Abhängigkeiten der Systemkomponenten (12)

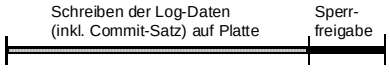
- Commit-Behandlung (Forts.)
  - Gruppen-Commit
    - Log-Datei ist potentieller Leistungsengpass
      - pro Änderungstransaktion wenigstens 1 Log-E/A
      - max. ca. 250 sequentielle Schreibvorgänge pro Sekunde (1 Platte)
    - Gruppen-Commit bedeutet gemeinsames Schreiben der Log-Daten von mehreren TA
      - 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
      - nur geringe Commit-Verzögerung

Abhängigkeiten der Systemkomponenten (13)

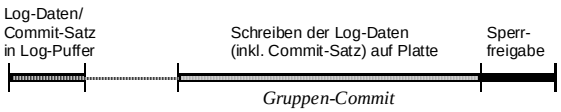
- Commit-Behandlung (Forts.)
  - Gruppen-Commit (Forts.)
    - Gruppen-Commit erlaubt Reduktion auf 0.1 - 0.2 Log-E/As pro TA
      - Einsparung an CPU-Overhead für E/A reduziert CPU-Wartezeiten
      - dynamische Festsetzung des Timer-Wertes durch DBVS wünschenswert
    - Gruppen-Commit ermöglicht demnach Durchsatzverbesserung v.a. bei Log-Engpass oder hoher CPU-Auslastung

Abhängigkeiten der Systemkomponenten (14)

- Commit-Behandlung (Forts.)
  - Vergleich verschiedener Commit-Verfahren
    - Standard-2PC

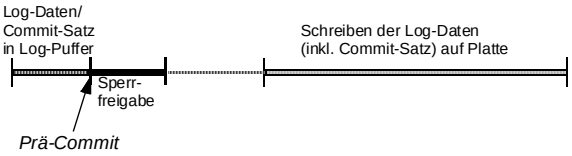


    - Gruppen-Commit



Abhängigkeiten der Systemkomponenten (15)

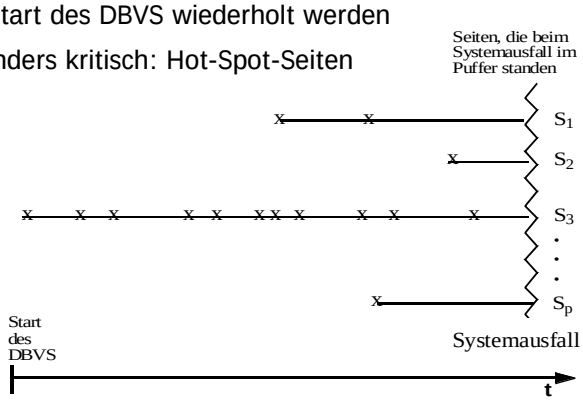
- Commit-Behandlung (Forts.)
  - Vergleich verschiedener Commit-Verfahren (Forts.)
    - Weitere Optimierungsmöglichkeit: *Prä-Commit*
      - Sperren bereits freigeben, wenn Commit-Satz im Log-Puffer steht (vor Schreiben auf Log-Platte)
      - TA kann nur noch durch Systemfehler scheitern
      - In diesem Fall scheitern auch alle ‚abhängigen' TA, die ungesicherte Änderungen aufgrund der vorzeitigen Sperrfreigabe gesehen haben



      - In allen drei Verfahren wird der Benutzer erst nach Schreiben des Commit-Satzes auf Platte vom TA-Ende informiert

Sicherungspunkte (1)

- 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 (2)

- 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 (3)

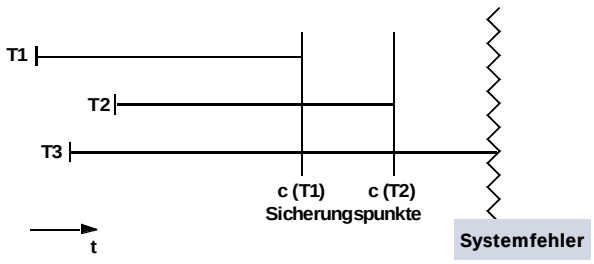
- Arten von Sicherungspunkten
  - Direkte Sicherungspunkte
    - alle geänderten Seiten im DB-Puffer werden in die permanente DB eingebracht
    - Redo-Recovery beginnt bei letztem Sicherungspunkt
    - Nachteil: lange ‚Totzeit' des Systems, da während des Sicherungspunktes keine Änderungen durchgeführt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

Sicherungspunkte (4)

- 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 (5)

- 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*)

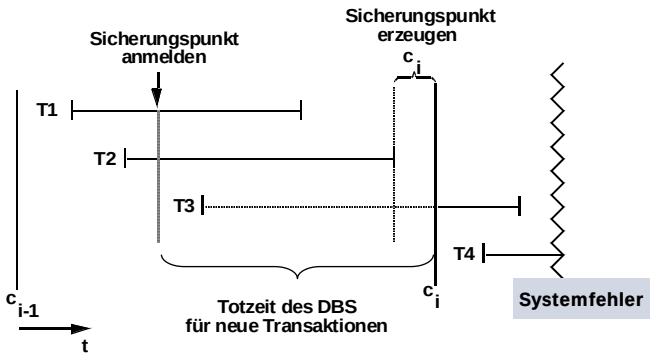


Sicherungspunkte (6)

- 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 (7)

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

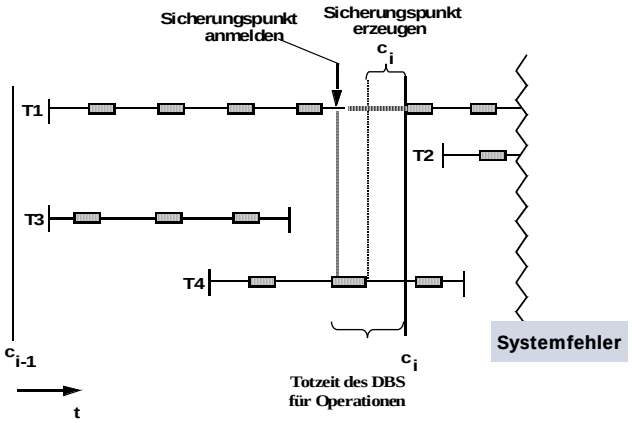


Sicherungspunkte (8)

- 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 (9)

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



Sicherungspunkte (10)

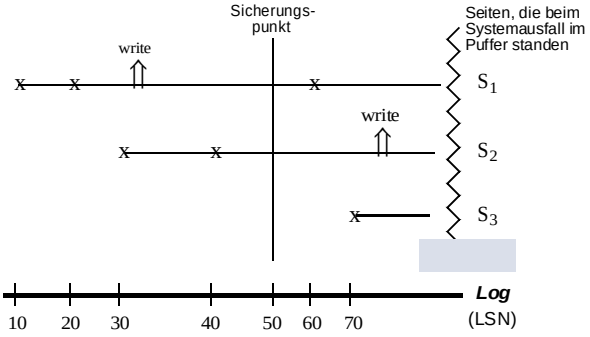
- Aktionskonsistente Sicherungspunkte (Forts.)
  - Eigenschaften
    - keine Änderungsanweisungen während des Sicherungspunktes
    - geringere Totzeiten als bei TCC, dafür Verminderung der Qualität der Sicherungspunkte
    - Crash-Recovery wird nicht durch letzten Sicherungspunkt begrenzt
  - Abhängigkeit: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

Sicherungspunkte (11)

- 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 (12)

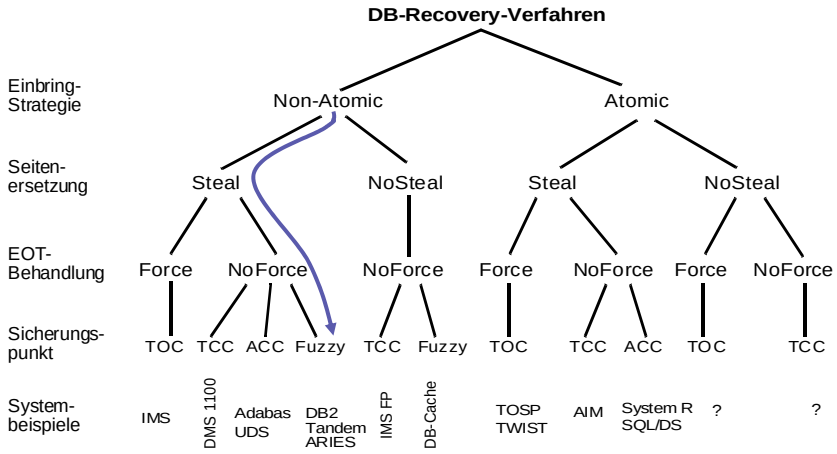
- Unscharfe Sicherungspunkte (Forts)



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

Sicherungspunkte (13)

- Kombinierbarkeit der Verfahren



Recovery (1)

- 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**

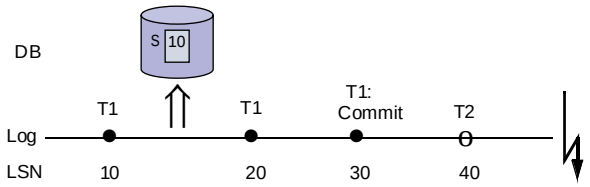
  - Mögliche Antworten:
    a. tu' nichts
    b. benutze die UNDO-Information und setze zurück
    c. benutze die REDO-Information und wiederhole
    d. WAL-Prinzip verhindert diese Situation
    e. 2PC verhindert diese Situation

Recovery (2)

- 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 (3)

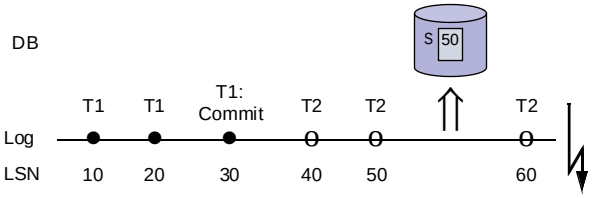
- Nutzung von LSNs (Forts.)
  - Vereinfachte Anwendung: Seitensperren



- 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 (4)

- 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 (4)

- 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 (5)

- 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 (6)

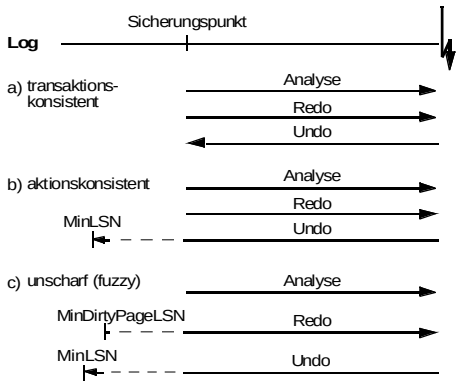
- Crash-Recovery (Forts.)
  - bei *atomic*
    - permanente DB entspricht Zustand des letzten erfolgreichen Einbringens (Sicherungspunkt)
    - zumindest aktionskonsistent → DML-Befehle ausführbar (logisches Logging)
    - *force:* kein Redo
    - *noforce:*
      - transaktionskonsistentes Einbringen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

Recovery (7)

- 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 (8)

- Allgemeine Restart-Prozedur (Forts.)

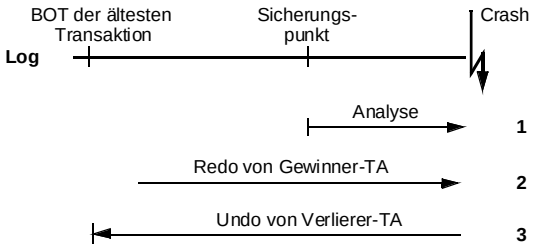


Recovery (9)

- 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 (10)

- 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 (11)

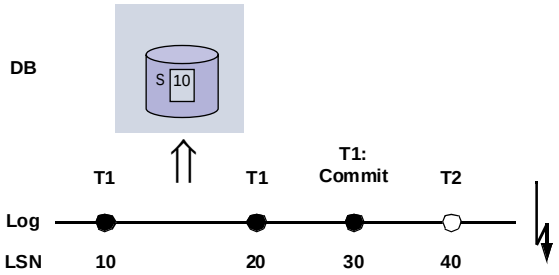
- Redo-Recovery
  - Notwendigkeit
    - bei physischem und physiologischem Logging
      - Redo-Aktion für Log-Satz L wird über PageLSN der betroffenen Seite B angezeigt

```
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 bei Crashs während des Restart?

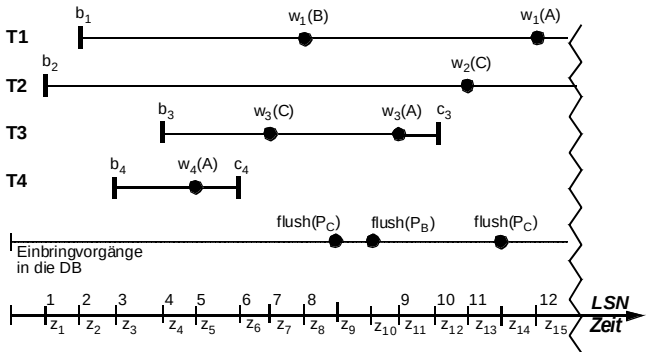
Recovery (12)

- Redo-Recovery (Forts.)



Recovery (13)

- Restart-Beispiel



Recovery (14)

- Restart-Beispiel (Forts.)

Zeit	Aktion	Änderung im DB-Puffer (Seite, LSN)	Änderung in der DB (Seite, LSN)	Log-Puffer: (LSN, TAID, Log-Info, PrevLSN)	Log-Datei: zugefügte Einträge (LSNs)
z_1	b_2			1, T_2, BOT, 0	
z_2	b_1			2, T_1, BOT, 0	
z_3	b_4			3, T_4, BOT, 0	
z_4	b_3			4, T_3, BOT, 0	
z_5	$w_4(A)$	P_A, 5		5, T_4, U/R(A), 3	
z_6	c_4			6, T_4, EOT, 5	1, 2, 3, 4, 5, 6
z_7	$w_3(C)$	P_C, 7		7, T_3, U/R(C), 4	
z_8	$w_1(B)$	P_B, 8		8, T_1, U/R(B), 2	
z_9	flush(P_C)		P_C, 7		7, 8
z_{10}	flush(P_B)		P_B, 8		
z_{11}	$w_3(A)$	P_A, 9		9, T_3, U/R(A), 7	
z_{12}	c_3			10, T_3, EOT, 9	9, 10
z_{13}	$w_2(C)$	P_C, 11		11, T_2, U/R(C), 1	
z_{14}	flush(P_C)		P_C, 11		11
z_{15}	$w_1(A)$	P_A, 12		12, T_1, U/R(A), 8	

Recovery (15)

- Restart-Beispiel (Forts.)
  - Annahme
    - zu Beginn seien alle Seiten-LSNs 0
  - Analyse-Phase:
    - Gewinner-TA: T3, T4
      Verlierer-TA: T1, T2
      relevante Seiten: P_A, P_B, P_C
  - Anmerkung
    - im Beispiel ändert nie mehr als eine TA gleichzeitig in einer Seite, was einem Einsatz von Seitensperren entspricht; deshalb ist Selektives Redo, also nur das Redo der Gewinner-TA, ausreichend

Recovery (16)

- Restart-Beispiel (Forts.)
  - Redo-Phase:
    - Log-Sätze für T_3 und T_4 vorwärts prüfen

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_4	P_A	$0 \rightarrow 5$	5	REDO
T_3	P_C	11	7	Kein REDO
T_3	P_A	$5 \rightarrow 9$	9	REDO

      - Redo nur, wenn Seiten-LSN < Log-Satz-LSN
      - Seiten-LSNs wachsen monoton

Recovery (17)

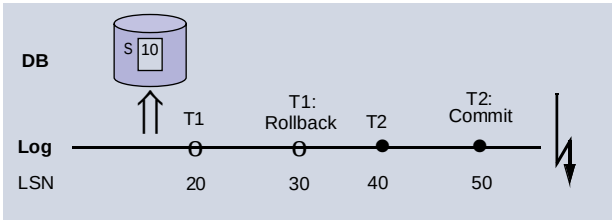
- Restart-Beispiel (Forts.)
  - Undo-Phase:
    - Log-Sätze für T_1 und T_2 rückwärts prüfen

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_1	P_A	9	12	Kein Undo, ohnehin nicht in Log-Datei
T_2	P_C	11	11	Undo
T_1	P_B	8	8	Undo

      - Undo nur, wenn Seiten-LSN ≥ Log-Satz-LSN
      - wegen der Seitensperren gibt es auf einer Seite keine Interferenz zwischen Redo- und Undo-Aktionen; Zustands-Logging sichert Undo-Idempotenz!

Recovery (18)

- Probleme bei LSN-Verwendung für Undo
  - Problem 1: Rücksetzungen von TA
    - bisherige LSN-Verwendung führt zu Problemen in der Undo-Phase bei vorherigem Rollback



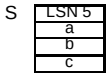
    - Redo von T2: S(10) < T2(40) : Redo, S(40)
    - Undo von T1: S(40) > T1(20) : Undo, Fehler

Recovery (19)

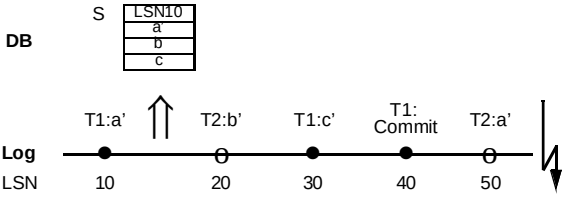
- Probleme bei LSN-Verwendung für Undo (Forts.)
  - Problem 1: Rücksetzungen von TA (Forts.)
    - Bemerkung
      - es wird Änderung 20 zurückgesetzt, obwohl sie gar nicht in der Seite S vorliegt
      - Zuweisung von LSN = 20 zu S verletzt Monotonieforderung für Seiten-LSNs
      - was passiert bei Crash nach Zuweisung?

Recovery (20)

- Probleme bei LSN-Verwendung für Undo (Forts.)
  - Problem 2: Satzsperren
    - Ausgangszustand der Seite S



    - T1 und T2 ändern gleichzeitig in Seite



      - Redo von T1: S(10) ≥ T1(10): kein Redo
        S(10) < T1(30): Redo, S(30)
      - Undo von T2 (LIFO): S(30) < T2(50): kein Undo
        S(30) > T2(20): Undo, Fehler!
    - Allgemeinere Behandlung des Undo erforderlich!

Recovery (21)

- Fehlertoleranz des Restart
  - Forderung: Idempotenz des Restart

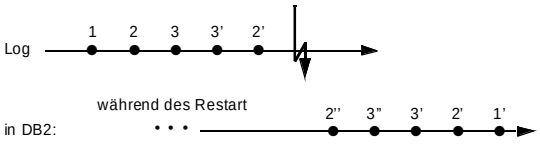
$$Undo(Undo(\ldots(Undo(A))\ldots)) = Undo(A)$$

$$Redo(Redo(\ldots(Redo(A))\ldots)) = Redo(A)$$

    - Idempotenz der Redo-Phase wird dadurch erreicht, dass LSN des Log-Satzes, für den ein Redo tatsächlich ausgeführt wird, in die Seite eingetragen wird; Redo-Operationen erfordern keine zusätzliche Protokollierung
    - Seiten-LSNs müssen monoton wachsen; deshalb kann in der Undo-Phase nicht entsprechend verfahren werden

Recovery (22)

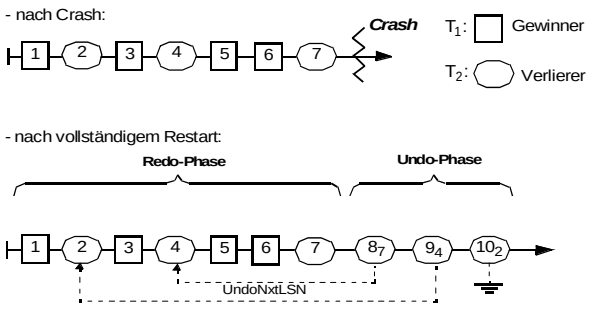
- Fehlertoleranz des Restart (Forts.)
  - Forderung: Idempotenz des Restart (Forts.)
    - Gewährleistung der Idempotenz der Undo-Phase erfordert ein neues Konzept: CLR = Compensation Log Record
      - Änderungen der DB sind durch Log-Einträge abzusichern - und zwar im Normalbetrieb und beim Restart!
      - was passiert im Fall eines Crash beim Undo?
        Aktionen 1-3 sollen zurückgesetzt werden:
        I' ist CLR für I und I'' ist CLR für I'



      - Problem von kompensierenden Kompensationen!
      - Crash bei Restart!?

Recovery (23)

- Compensation Log Records (CLR)
  - 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")
    - schematische Darstellung der Log-Datei

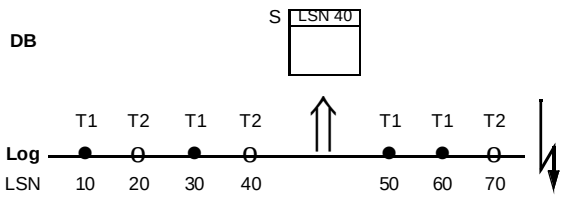


Recovery (24)

- 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)

Recovery (25)

- Compensation Log Records (Forts.)
  - Detaillierung des Beispiels
    - alle Änderungen betreffen Seite S
    - Zustand nach Crash 1

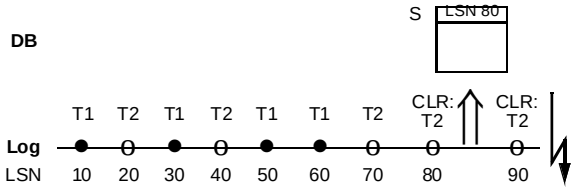


Recovery (26)

- Compensation Log Records (Forts.)
  - Fortsetzung des Beispiels (Forts.)
    - Zustand nach Crash 1 (Forts.)
      - repeating history:
        S(40) >T1(10): –
        . . .
        S(40) ≥T2(40): –
        S(40) < T1(50): Redo, S(50)
        S(50) <T1(60): Redo, S(60)
        S(60) < T2(70): Redo, S(70)
      - Undo von T2:
        CLR(80): Kompensieren von T2(70), S(80)
        Schreiben von S in die DB (Flush S)
        CLR(90): Kompensieren von T2(40), S(90)
        Crash

Recovery (27)

- Compensation Log Records (Forts.)
  - Fortsetzung des Beispiels (Forts.)
    - Zustand nach Crash 2



Recovery (28)

- Compensation Log Records (Forts.)
  - Fortsetzung des Beispiels (Forts.)
    - Zustand nach Crash 2 (Forts.)
      - Repeating History:

        S(80) >T1(10): –
        . . .
        S(80) > T2(70): –
        CLR(80): –
        CRL(90): Kompensieren von T2(40), S(90)
      - Undo von T2:

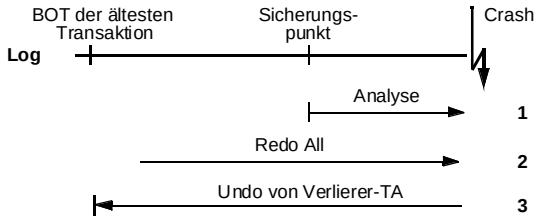
        CLR(100): Kompensieren von T2(20), S(100)
        Ende

Recovery (29)

- Restart-Prozedur bei Update-in-Place
  - Attribute: non-atomic, steal, noforce, fuzzy checkpoints
    1. Analyse-Phase
       - vom letzten Sicherungspunkt bis zum Log-Ende
    2. Redo-Phase
       - Startpunkt abhängig vom Sicherungspunkt-Typ: hier MinDirtyPageLSN
       - Vollständiges Redo oder Repeating History: Wiederholung aller Änderungen (auch von Verlierer-TA), falls erforderlich
    3. Undo-Phase
       - Rücksetzen der Verlierer-TA bis MinLSN

Recovery (30)

- Restart-Prozedur bei Update-in-Place (Forts.)
  - Attribute: non-atomic, steal, noforce, fuzzy checkpoints (Forts.)

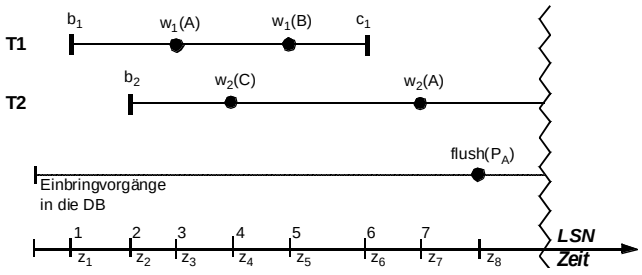


    - Umsetzung durch ARIES
      - Algorithm for Recovery and Isolation Exploiting Semantics
      - entwickelt von C. Mohan et al. (IBM Almaden Research)
      - realisiert in einer Reihe von kommerziellen DBS

Mohan, C. et al.: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, in ACM TODS 17:1, 1992, 94-162

Recovery (31)

- Restart-Beispiel 2



Recovery (32)

- Restart-Beispiel 2 (Forts.)

Zeit	Aktion	Änderung im DB-Puffer (Seite, LSN)	Änderung in der DB (Seite, LSN)	Log-Puffer: (LSN, TAID, Log-Info, PrevLSN)	Log-Datei: zugefügte Ein-träge (LSNs)
z_1	b_1			1, T_1, BOT, 0	
z_2	b_2			2, T_2, BOT, 0	
z_3	$w_1(A)$	P_A, 3		3, T_1, U/R(A), 1	
z_4	$w_2(C)$	P_C, 4		4, T_2, U/R(C), 2	
z_5	$w_1(B)$	P_B, 5		5, T_1, U/R(B), 3	
z_6	c_1			6, T_1, EOT, 5	1, 2, 3, 4, 5, 6
z_7	$w_2(A)$	P_A, 7		7, T_2, U/R(A), 4	
z_8	flush(P_A)		P_A, 7		7

Recovery (33)

- Restart-Beispiel 2 (Forts.)
  - Annahme
    - zu Beginn seien alle Seiten-LSNs 0
  - Analyse-Phase
    - Gewinner-TA: T_1
      Verlierer-TA: T_2
      relevante Seiten: P_A, P_B, P_C
  - Anmerkung
    - im Restart-Beispiel 2 wird vollständiges Redo durchgeführt; zur Gewährleistung der Idempotenz der Undo-Operationen wird für jede ausgeführte Undo-Operation ein CLR mit folgender Struktur angelegt
      - [LSN, TAID, PageID, Redo, PrevLSN, UndoNextLSN]

Recovery (34)

- Restart-Beispiel 2 (Forts.)
  - Redo-Phase
    - Log-Sätze aller TA (T_1, T_2) vorwärts prüfen

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_1	P_A	7	3	Kein REDO
T_2	P_C	$0 \rightarrow 4$	4	REDO
T_1	P_B	$0 \rightarrow 5$	5	REDO
T_2	P_A	7	7	Kein REDO

      - Redo nur, wenn Seiten-LSN < Log-Satz-LSN

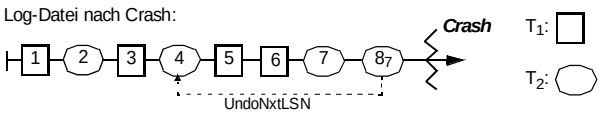
Recovery (35)

- Restart-Beispiel 2 (Forts.)
  - Undo-Phase
    - Log-Sätze der Verlierer-TA T_2 rückwärts unabhängig von Seiten-LSN prüfen
      - für jeden Log-Satz wird die zugehörige Undo-Operation durchgeführt und mit einem CLR in der Log-Datei vermerkt

TA	Log-Satz-LSN	Aktion
T_2	7	UNDO und lege CLR[8, T_2, P_A, U(A), 7, 4] an
T_2	4	UNDO und lege CLR[9, T_2, P_C, U(C), 8, 2] an
T_2	2	UNDO und lege CLR[10, T_2, _ , _ , 9, 0] an

Recovery (36)

- Restart-Beispiel 2 (Forts.)
  - Annahme
    - Crash während des Restart



  - Analyse-Phase
    - dto
  - Redo-Phase
    - Log-Sätze aller TA (T_1, T_2) inkl. CLRs vorwärts prüfen
    - für jedes CLR wird jeweils Redo ausgeführt

Recovery (37)

- Restart-Beispiel 2 (Forts.)
  - Redo-Phase
    - Log-Sätze aller TA (T_1, T_2) inkl. CLRs vorwärts prüfen
    - für jedes CLR wird jeweils Redo ausgeführt

TA	Seite	Seiten-LSN	Log-Satz-LSN	Aktion
T_1	P_A	7	3	Kein REDO
T_2	P_C	4	4	Kein REDO
T_1	P_B	5	5	Kein REDO
T_2	P_A	7	7	Kein REDO
T_2	P_A	$7 \rightarrow 8$	8	REDO: U(A)

Recovery (38)

- Restart-Beispiel 2 (Forts.)
  - Undo-Phase
    - Log-Sätze der Verlierer-TA T_2 (inkl. CLRs) rückwärts unabhängig von Seiten-LSN prüfen
    - für jeden Log-Satz wird die zugehörige Undo-Operation durchgeführt und mit einem CLR in der Log-Datei vermerkt

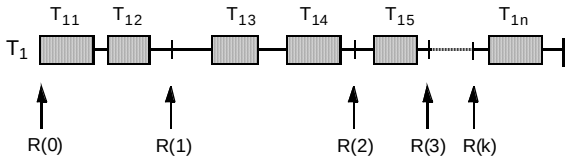
TA	Log-Satz-LSN	Aktion
T_2	8	UndoNxtLSN = 4, dann weiter mit 4. Log-Satz (7. Log-Satz wird übersprungen, da er bereits mit dem 8. kompensiert wurde)
T_2	4	UNDO und lege CLR[9, T_2, P_C, U(C), 8, 2] an
T_2	2	UNDO und lege CLR[10, T_2, _ , _ , 9, 0] an

Recovery (39)

- Zurücksetzen von Transaktionen
  - Transaktions-Recovery
    - Zurücksetzen einer TA im laufenden DB-Betrieb
    - Nutzung der PrevLSN-Kette im temporären Log
    - Schreiben von optimierten CLRs, um mehrfaches Rücksetzen bei Restart zu vermeiden
  - Erweiterung zum partiellen Zurücksetzen
    - Voraussetzung
      - transaktionsinterne Rücksetzpunkte (*Savepoints*)
    - Zusätzliche Operationen: Save R(i), Restore R(j)
    - Protokollierung aller Änderungen, Sperren, Cursor-Positionen usw.
    - Undo-Operation bis R(j) in LIFO-Reihenfolge

Recovery (40)

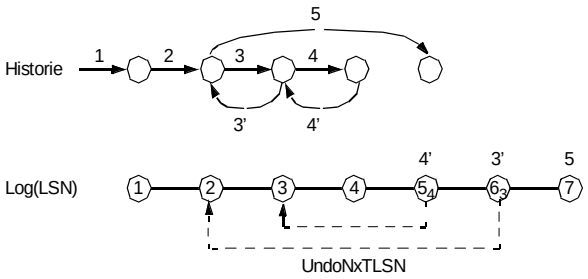
- Zurücksetzen von Transaktionen (Forts.)
  - Erweiterung zum partiellen Zurücksetzen (Forts.)



= atomarer Transaktionsschritt T_{1i}
R(i) = i-ter Rücksetzpunkt
$T_1 = (T_{11}, T_{12}, ..., T_{1n})$

Recovery (41)

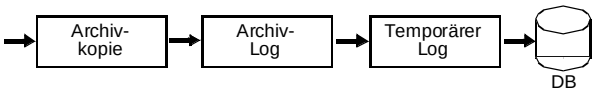
- Zurücksetzen von Transaktionen (Forts.)
  - Partielles Zurücksetzen einer TA



- Rücksetzpunkte müssen vom DBS sowie vom Laufzeitsystem der Programmiersprache unterstützt werden
  - derzeitige Implementierungen bieten keine Unterstützung von persistenten *Savepoints*!
  - nach Systemfehler wird TA vollständig zurückgesetzt

Recovery (42)

- Platten-Recovery
  - Spiegelplatten
    - schnellste und einfachste Lösung
    - hohe Speicherkosten
    - Doppelfehler nicht auszuschließen
  - Alternative
    - Archivkopie + Archiv-Log
    - sind längerfristig verfügbar zu halten (auf Band)
    - Problem von Alterungsfehlern
      - Führen von Generationen der Archivkopie
      - Duplex-Logging für Archiv-Log

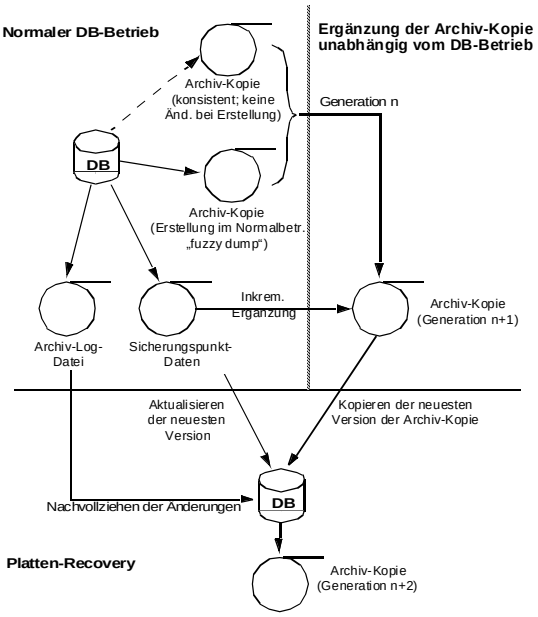


Recovery (43)

- Platten-Recovery (Forts.)
  - Ableitung von Archivdaten
    - Sammlung sehr großer Datenvolumina als nachgelagerter Prozess
    - Archiv-Log kann offline aus temporärer Log-Datei abgeleitet werden
    - Erstellung von Archivkopien und Archiv-Log erfolgt segmentorientiert
  - Optimierung der Erstellung der Archiv-Kopie durch inkrementelle Ergänzung mit Daten von Sicherungspunkten und ggf. Archiv-Log

Recovery (44)

- Platten-Recovery (Forts.)
  - Szenario/ Komponenten



Recovery (45)

- Erstellung der Archivkopie
  - Anhalten des Änderungsbetriebs zur Erstellung einer DB-Kopie i. allg. nicht tolerierbar
  - Alternativen:
    - incremental dumping (a)
    - Ableiten neuer Generationen aus 'Urkopie'
    - nur Änderungen seit der letzten Archiv-Kopie protokollieren
    - Offline-Erstellung einer aktuelleren Kopie
    - Online-Erstellung einer Archivkopie (b)
    - parallel zum Änderungsbetrieb

Recovery (46)

- Erstellung der Archivkopie (Forts.)
  - Unterschiedliche Konsistenzgrade:
    - fuzzy dump (b1)
      - Kopieren der DB im laufenden Betrieb, kurze Lesesperren
      - bei Plattenfehler Archiv-Log ab Beginn der Dump-Erstellung anzuwenden
    - aktionskonsistente Archivkopie (b2)
      - Voraussetzung bei logischem Operations-Logging
    - transaktionskonsistente Archivkopie (b3)
      - Voraussetzung bei logischem Transaktions-Logging
      - Black-/White-Verfahren
      - Copy-on-Update-Verfahren

Recovery (47)

- Black-/White-Verfahren
  - Ziel
    - Erzeugung transaktionskonsistenter Archiv-Kopien
  - spezieller Dumpprozess zur Erstellung der Archiv-Kopie
  - Kennzeichnung der Seiten
    - Paint-Bit (pro Seite)
      - weiß: Seite wurde noch nicht überprüft
      - schwarz: Seite wurde bereits verarbeitet
    - Modified-Bit (pro Seite)
      - zeigt an, ob eine Änderung seit Erstellung der letzten Archiv-Kopie erfolgte

C. Pu: On-the-Fly, Incremental, Consistent Reading of Entire Databases, in: Algorithmica, 1986, 271- 287

Recovery (48)

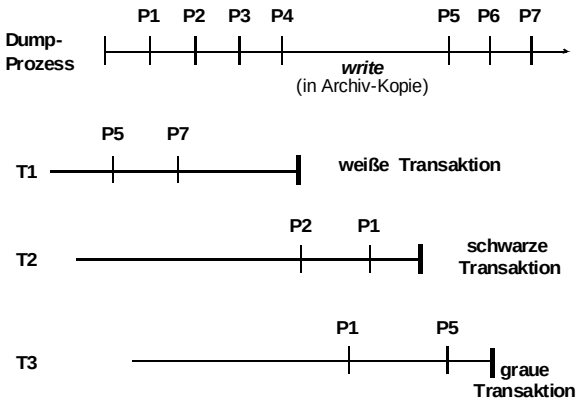
- Black-/White-Verfahren (Forts.)
  - Kennzeichnung der Seiten (Forts.)
    - Dumpprozess färbt alle weißen Seiten schwarz und schreibt geänderte Seiten in Archiv-Kopie:

```
WHILE there are white pages  DO;
      lock any white page;
      IF page is modified THEN DO;
          write page to archive copy;
          clear modified bit;
      END;
      change page color;
      release page lock;
  END;
```

  - Rücksetzregel
    - Transaktionen, die sowohl weiße als auch schwarze Objekte geändert haben ('graue Transaktionen'), werden zurückgesetzt
    - 'Farbtest' am Transaktionsende

Recovery (49)

- Black-/White-Verfahren (Forts.)
  - Beispiel



Recovery (50)

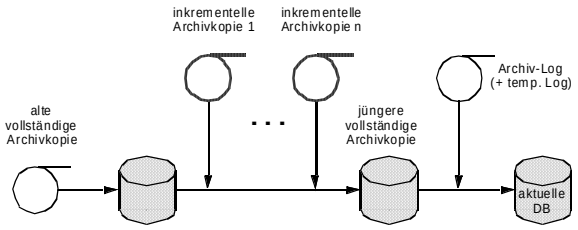
- Black-/White-Verfahren (Forts.)
  - Erweiterungen zur Vermeidung von Rücksetzungen
    - Turn-White-Strategien (Turn gray transactions white)
      - für graue Transaktionen werden Änderungen 'schwarzer' Objekte nachträglich in Archiv-Kopie geschrieben
      - Problem: transitive Abhängigkeiten
      - **Alternative:** alle Änderungen schwarzer Objekte seit Dump-Beginn werden noch geschrieben (repaint all)
      - Problem: Archiv-Kopie-Erstellung kommt u.U. nie zu Ende
    - Turn-Black-Strategien
      - Während der Erstellung einer Archiv-Kopie werden keine Zugriffe auf weiße Objekte vorgenommen
      - ggf. zu warten, bis Objekt gefärbt wird

Recovery (51)

- Black-/White-Verfahren (Forts.)
  - Erweiterungen zur Vermeidung von Rücksetzungen (Forts.)
    - Alternative: *Copy-on-Update* ("save some")
      - während der Erstellung einer Archiv-Kopie wird bei Änderung eines weißen Objektes Kopie mit Before-Image der Seite angelegt
      - Dump-Prozess greift auf Before-Images zu
      - Archiv-Kopie entspricht DB-Schnappschuss bei Dump-Beginn
      - wird in einigen DBS eingesetzt (DEC RDB)

Recovery (52)

- Inkrementelles Dumping
  - nur DB-Seiten, die seit der letzten Archivkopie-Erstellung geändert wurden, werden archiviert



  - Erkennung geänderter Seiten
    - Archivierungs-Bit pro Seite → sehr hoher E/A-Aufwand
    - besser: Verwendung separater Datenstrukturen (Bitlisten)
  - Setzen eines Änderungsbits falls
    - (PageLSN der ungeänderten Seite) < (LSN zu Beginn des letzten Dumps)

Zusammenfassung (1)

- Fehlerarten
  - Transaktions-, System-, Gerätefehler und Katastrophen
- Spektrum von Logging- und Recovery-Verfahren
  - Logging kann auf verschiedenen Systemebenen angesiedelt werden
  - erfordert ebenenspezifische Konsistenz im Fehlerfall
  - Eintrags-Logging ist Seiten-Logging überlegen
    - in vielen DBS findet sich das physiologische Logging (flexiblere Recovery in einer DB-Seite, geringerer Platzbedarf, weniger E/As, Gruppen-Commit)

Zusammenfassung (2)

- Zusammenspiel mit anderen Systemkomponenten
  - Synchronisationsgranulat muss größer oder gleich dem Log-Granulat sein
  - Atomic-Verfahren
    - erhalten den DB-Zustand des letzten Sicherungspunktes
    - gewährleisten demnach die gewählte Aktionskonsistenz auch bei der Recovery von einem Crash und
    - erlauben folglich logisches Logging
  - Update-in-Place-Verfahren
    - sind i. allg. Atomic-Strategien vorzuziehen, weil sie im Normalbetrieb wesentlich billiger sind und
    - nur eine geringe Crash-Wahrscheinlichkeit zu unterstellen ist
    - erfordern jedoch physisches Logging

Zusammenfassung (3)

- Grundprinzipien bei Update-in-Place
  - WAL-Prinzip: Write Ahead Log für Undo-Info
  - Redo-Info ist spätestens bei Commit zu schreiben
- Grundprinzipien bei Atomic
  - WAL-Prinzip bei verzögertem Einbringen:
    - TA-bezogene Undo-Info ist vor Sicherungspunkt zu schreiben
  - Redo-Info ist spätestens bei Commit auf die Log-Datei zu schreiben
- NoForce-Strategien
  - sind Force-Verfahren vorzuziehen
  - erfordern den Einsatz von Sicherungspunkt-Maßnahmen zur Begrenzung des Redo-Aufwandes
    - Fuzzy Checkpoints erzeugen den geringsten Overhead im Normalbetrieb

Zusammenfassung (4)

- Steal-Methoden
  - verlangen die Einhaltung des WAL-Prinzips
  - erfordern Undo-Aktionen nach einem Rechnerausfall
- Idempotenz des Restart
  - Operationen der Redo-Phase, falls erforderlich, erhöhen die Seiten-LSNs; Notwendigkeit der Wiederholung kann jederzeit erkannt werden
  - Idempotenz für Undo- und Rollback-Operationen durch Einführung von CLRs; nach Crash in der Undo-Phase werden Undo-Operationen beim nachfolgenden Restart in der Redo-Phase kompensiert (Erhöhung der Seiten-LSNs, beliebig oft unterbrechbar)
- Erstellung von Archiv-Kopien
  - *fuzzy dump* oder *copy on update* am geeignetsten