



Ludwig Maximilians Universität München  
Institut für Informatik  
Lehr- und Forschungseinheit für Datenbanksysteme

Datenbanksysteme II  
Kapitel 3: Logging & Recovery

Skript zur Vorlesung  
**Datenbanksysteme II**  
Sommersemester 2007

# Kapitel 3: Logging & Recovery

Vorlesung: Christian Böhm  
Übungen: Elke Achtert, Peter Kunath, Alexey Pryakhin

Skript © 2006 Christian Böhm

<http://www.dbs.informatik.uni-muenchen.de/Lehre/DBSII>



## Inhalt

Datenbanksysteme II  
Kapitel 3: Logging & Recovery

1. Recovery-Arten
2. Logging-Techniken
3. Abhängigkeiten zu anderen Systemkomponenten
4. Sicherungspunkte



## Transaktions-Recovery

### Transaktionsfehler

- Lokaler Fehler einer noch nicht festgeschriebenen TA, z.B. durch
  - Fehler im Anwendungsprogramm
  - Expliziter Abbruch der TA durch den Benutzer (ROLLBACK)
  - Verletzung von Integritätsbedingungen oder Zugriffsrechten
  - Rücksetzung aufgrund von Synchronisationskonflikten
- Behandlung durch **Rücksetzen**
  - *Lokales UNDO*: der ursprüngliche DB-Zustand wie zu BOT wird wiederhergestellt, d.h. Rücksetzen aller Aktionen, die diese TA ausgeführt hat
  - Transaktionsfehler treten relativ häufig auf  
→ Behebung innerhalb von Millisekunden notwendig



## Crash Recovery

### Systemfehler

- Fehler mit Hauptspeicherverlust, d.h. permanente Speicher sind *nicht* betroffen, z.B. durch
  - Stromausfall
  - Ausfall der CPU
  - Absturz des Betriebssystems, ...
- Behandlung durch **Crash Recovery** (Warmstart)
  - *Globales UNDO*: Rücksetzen aller noch nicht abgeschlossenen TAs, die **bereits** in die DB eingebracht wurden
  - *Globales REDO*: Nachführen aller bereits abgeschlossenen TAs, die **noch nicht** in die DB eingebracht wurden
  - Systemfehler treten i.d.R. im Intervall von Tagen auf  
→ Recoverydauer einige Minuten



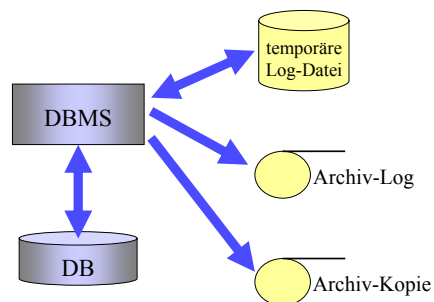
## Geräte-Recovery

### Medienfehler

- Fehler mit Hintergrundspeicherverlust, d.h. Verlust von permanenten Daten, z.B. durch
  - Plattencrash
  - Brand, Wasserschaden, ...
  - Fehler in Systemprogrammen, die zu einem Datenverlust führen
- Behandlung durch **Geräte-Recovery** (Kaltstart)
  - Aufsetzen auf einem früheren, gesicherten DB-Zustand (Archivkopie)
  - *Globales REDO*: Nachführen aller TAs, die nach dem Erzeugen der Sicherheitskopie abgeschlossen wurden
  - Medienfehler treten eher selten auf (mehrere Jahre)
    - Recoverydauer einige Stunden / Tage
  - Wichtig: regelmäßige Sicherungskopien der DB notwendig



## Systemkomponenten der DB-Recovery



- Behandlung von Transaktions- und Systemfehlern  
 $DB + \text{temporäre Log-Datei} \rightarrow DB$
- Behandlung von Medienfehlern  
 $Archiv-Kopie + Archiv-Log \rightarrow DB$



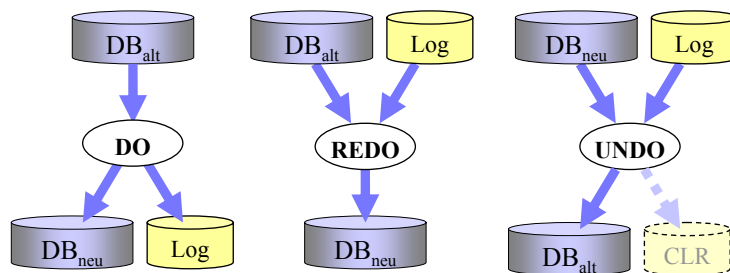
## Inhalt

1. Recovery-Arten
2. Logging-Techniken
3. Abhängigkeiten zu anderen Systemkomponenten
4. Sicherungspunkte



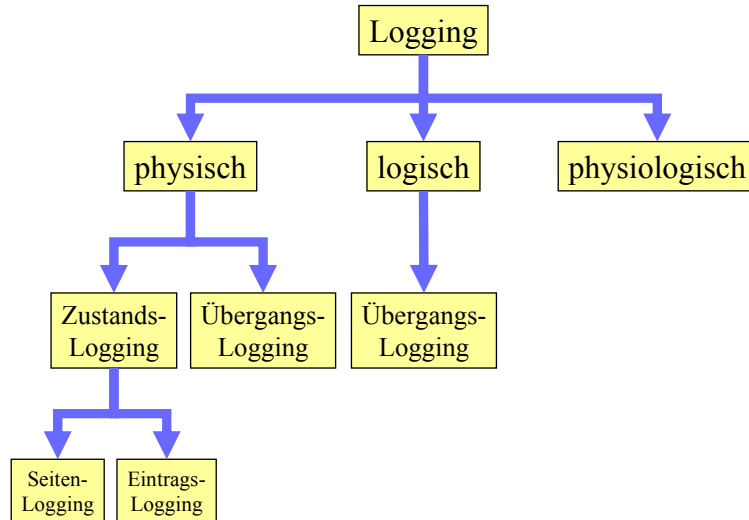
## Aufgaben des Logging

- Für jede Änderungsoperation auf der Datenbank im Normalbetrieb (**DO**) benötigt man Protokolleinträge für
  - **REDO**: Information zum Nachvollziehen der Änderungen erfolgreicher TAs
  - **UNDO**: Information zum Zurücknehmen der Änderungen unvollständiger TAs





## Klassifikation von Logging-Verfahren



## Physisches Logging (1)

- Protokoll auf der Ebene der physischen Objekte (Seiten, Datensätze, Indexeinträge)
- **Zustandslogging**
  - Protokollierung der Werte vor und nach jeder Änderung:
  - Alte Zustände *BFIM* (Before-Images) und neue Zustände *AFIM* (After-Images) der geänderten Objekte werden in die Log-Datei geschrieben
- **Zustandslogging auf Seitenebene**
  - vollständige Kopien von Seiten werden protokolliert
  - Recovery sehr einfach und schnell, da Seiten einfach zurückkopiert werden
  - sehr großer Logumfang und hohe I/O-Kosten auch bei nur kleinen Änderungen
  - Seitenlogging impliziert Seitensperren → hohe Konfliktrate bei Synchronisation



## Physisches Logging (2)

- **Zustandslogging auf Eintragebene**
  - statt ganzer Seiten werden nur tatsächlich geänderte Einträge protokolliert
  - kleinere Sperrgranulate als Seiten möglich
  - Protokollgröße reduziert sich typischerweise um mindestens eine Größenordnung
  - Log-Einträge werden in Puffer gesammelt → wesentlich weniger Plattenzugriffe
  - Recovery ist aufwändiger: zu ändernde Datenbankseiten müssen vollständig in den Hauptspeicher geladen werden, um die Log-Einträge anwenden zu können



## Physisches Logging (3)

- **Übergangslogging**
  - Protokollierung der Zustandsdifferenz zwischen *BFIM* und *AFIM*
  - Aus *BFIM* muss *AFIM* berechenbar sein (u.u.)
  - Realisierbar durch *XOR*-Operation  $\oplus$  (eXclusive-OR):

	Zustands-Logging	Übergangs-Logging
<b>DO</b> Änderung $A_{alt} \rightarrow A_{neu}$	Protokollierung von $BFIM = A_{alt}, AFIM = A_{neu}$	Protokollierung von $D = A_{alt} \oplus A_{neu}$
<b>REDO</b> (in DB liegt $A_{alt}$ )	Überschreibe $A_{alt}$ mit <i>AFIM</i>	$A_{neu} = A_{alt} \oplus D$
<b>UNDO</b> (in DB liegt $A_{neu}$ )	Überschreibe $A_{neu}$ mit <i>BFIM</i>	$A_{alt} = A_{neu} \oplus D$

*XOR*:  
 $0 \oplus 0 = 0$   
 $0 \oplus 1 = 1$   
 $1 \oplus 0 = 1$   
 $1 \oplus 1 = 0$



## Logisches Logging (1)

- Spezielle Form des Übergangs-Logging: nicht physische Zustandsänderungen protokollieren, sondern Änderungsoperationen mit ihren aktuellen Parametern
- **Vorteil:** Protokoll auf hoher Abstraktionsebene ermöglicht kurze Log-Einträge
- **Probleme für REDO**  
Änderungen umfassen typischerweise mehrere Seiten (Tabelle, Indexe)
  - Atomares Einbringen der Mehrfachänderungen schwierig.
  - Logische Änderungen sind aufwändiger durchzuführen als physische Änderungen



## Logisches Logging (2)

- **Probleme für UNDO**  
Mengenorientierte Änderungen können sehr aufwändige Protokolleinträge verursachen:
  - Bsp. `DELETE FROM Products WHERE Group = 'G1'`  
=> *UNDO* erfordert viele Einfügungen, falls Produktgruppe G1 umfangreich ist
  - Bsp. `UPDATE Products SET Group = 'G2' WHERE Group = 'G1'`  
=> *UNDO* muss alte und neue Produkte der Gruppe G2 unterscheiden



## Physiologisches Logging

- Kombination von physischem und logischem Logging:  
Protokollierung von *elementaren Operationen innerhalb einer Seite*
- **Physical-to-a-page**
  - Protokollierungseinheiten sind geänderte Seiten
  - gut verträglich mit Pufferverwaltung und direktem (atomarem) Einbringen
- **Logical-within-a-page**
  - logische Protokollierung der Änderungen auf einer Seite
- **Bewertung**
  - Log-Einträge beziehen sich nicht auf mehrere Seiten wie bei logischem Logging
  - Dadurch einfachere Recovery als bei logischem Logging
  - Log-Datei ist länger als bei logischem Logging aber kürzer als bei physischem Logging
  - Flexibler als physisches Logging wegen variabler Objektpositionen auf Seiten.



## Aufbau der Log-Datei (1)

- **Art der Protokolleinträge**
  - Beginn, Commit und Rollback von Transaktionen
  - Änderungen des DB-Zustandes durch Transaktionen
  - Sicherungspunkte (Checkpoints)
- **Struktur der Log-Einträge für Änderungen**  
(*LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN*)
  - *LSN (Log Sequence Number)*: eindeutige Kennung des Log-Eintrags in chronologischer Reihenfolge
  - *TA-Id*: eindeutige Kennung der TA, die die Änderung durchgeführt hat
  - *Page-Id*: Kennung der Seite auf der die Änderungsoperation vollzogen wurde (ein Eintrag pro geänderter Seite)
  - *REDO*: gibt an, wie die Änderung nachvollzogen werden kann
  - *UNDO*: beschreibt, wie die Änderung rückgängig gemacht werden kann
  - *PrevLSN*: Zeiger auf vorhergehenden Log-Eintrag der jeweiligen TA (Effizienzgründe)





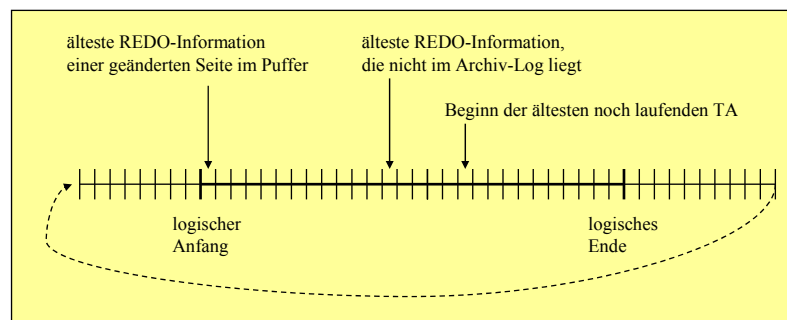
## Aufbau der Log-Datei (2)

- 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 TAs werden nicht mehr benötigt
  - Nach Einbringen der Seite in die DB wird *REDO*-Information nicht mehr benötigt
- *REDO*-Information für **Geräte-Recovery** ist im Archiv-Log zu sammeln



## Aufbau der Log-Datei (3)

- **Ringpuffer-Organisation** der Log-Datei





## Beispiel einer Log-Datei

Ablauf T <sub>1</sub>	Ablauf T <sub>2</sub>	Log-Eintrag <i>(LSN, TA-Id, Page-Id, REDO, UNDO, PrevLSN)</i>
<b>begin</b>		(#1, T <sub>1</sub> , begin, 0)
read(A, a <sub>1</sub> )		(#2, T <sub>2</sub> , begin, 0)
 a <sub>1</sub> := a <sub>1</sub> - 50 <b>write</b> (A, a <sub>1</sub> )	<b>begin</b> read(C, c <sub>2</sub> ) //80	(#3, T <sub>1</sub> , p <sub>A</sub> , A-=50, A+=50, #1)
 read(B, b <sub>1</sub> ) //70 b <sub>1</sub> := 50 <b>write</b> (B, b <sub>1</sub> ) <b>commit</b>	 c <sub>2</sub> := 100 <b>write</b> (C, c <sub>2</sub> )	(#4, T <sub>2</sub> , p <sub>C</sub> , C=100, C=80, #2)
	 read(A, a <sub>2</sub> ) a <sub>2</sub> := a <sub>2</sub> - 100 <b>write</b> (A, a <sub>2</sub> ) <b>commit</b>	(#5, T <sub>1</sub> , p <sub>B</sub> , B=50, B=70, #3) (#6, T <sub>1</sub> , commit, #5)
		(#7, T <sub>2</sub> , p <sub>A</sub> , A-=100, A+=100, #4) (#8, T <sub>2</sub> , commit, #7)

19

(hier: logisches Logging)



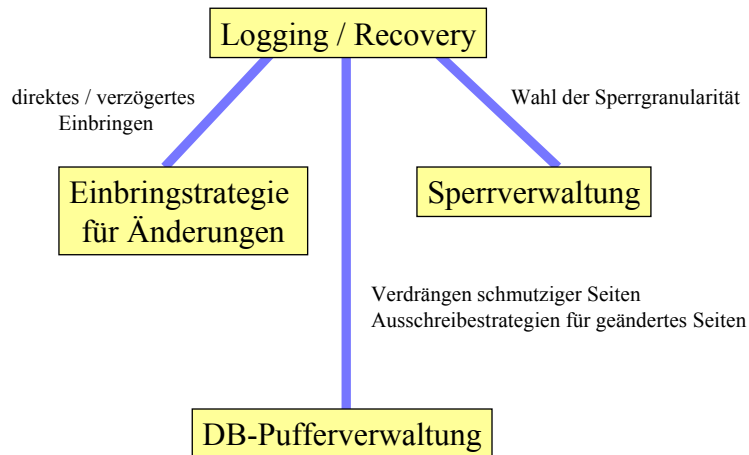
## Inhalt

1. Recovery-Arten
2. Logging-Techniken
3. Abhängigkeiten zu anderen Systemkomponenten
4. Sicherungspunkte

20



## Abhängigkeiten zu anderen Systemkomponenten



## Einbringstrategien (1)

- **Direktes Einbringen (*NonAtomic, Update-in-Place*)**
  - Geänderte Objekte werden immer auf denselben Block auf Platte zurück geschrieben
  - Schreiben ist dadurch gleichzeitig Einbringen in die permanente DB
  - Es ist nicht möglich mehrere Seiten *atomic* einzubringen, d.h. Unterbrechungsfreiheit des Einbringens kann nicht garantiert werden (*NonAtomic*).



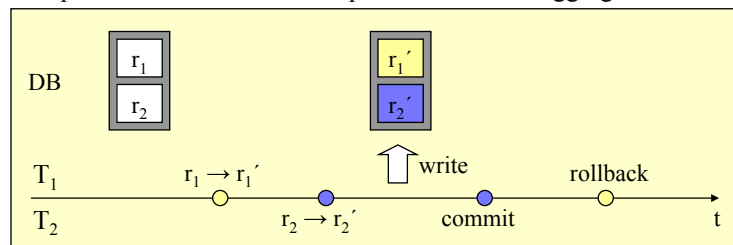
## Einbringstrategien (2)

- **Indirektes (verzögertes) Einbringen (*Atomic*)**
  - Ziel: Einbringen in die Datenbank wird unterbrechungsfrei durchgeführt
  - Geänderte Seite wird in separaten Block auf Platte geschrieben (z.B. *Schattenspeichertechnik* oder *Twin-Block-Verfahren*)
  - Einbringen in die DB kann von *COMMIT* losgelöst werden und z.B. erst beim nächsten Sicherungspunkt stattfinden (*verzögertes Einbringen*)
  - Atomares Einbringen mehrerer geänderter Seiten ist durch Umschalten von Seitentabellen möglich (*Atomic*)
  - Alte Versionen der Objekte bleiben erhalten, d.h. es muss keine *UNDO*-Information explizit gespeichert werden



## Einfluss der Sperrgranularität

- Log-Granularität muss kleiner oder gleich der Sperrgranularität sein, sonst Lost Updates möglich
- D.h. Satzsperrn erzwingen feine Log-Granulate
- Beispiel für Problem bei “Satzsperrn mit Seitenlogging”



- T<sub>1</sub> und T<sub>2</sub> ändern die Datensätze r<sub>1</sub> und r<sub>2</sub>, die auf derselben DB-Seite liegen
- Die Seite wird in die DB zurück geschrieben, T<sub>2</sub> endet mit *COMMIT*
- Falls T<sub>1</sub> zurückgesetzt wird, geht auch die Änderung r<sub>2</sub> → r<sub>2</sub>' verloren
- Lost Update, d.h. eklatanter Verstoß gegen die Dauerhaftigkeit des *COMMIT*



## Pufferverwaltung (1)

- **Verdrängungsstrategien**

Ersetzung „schmutziger“ Seiten im Puffer, d.h.  $\text{Seite}_{\text{Puffer}} \neq \text{Seite}_{\text{DB}}$

- **No-Steal**

- Schmutzige Seiten dürfen nicht aus dem Puffer entfernt werden
- DB enthält keine Änderungen nicht-erfolgreicher TAs
- *UNDO*-Recovery ist nicht erforderlich
- Probleme bei langen Änderungs-TAs, da große Teile des Puffers blockiert werden

- **Steal**

- Schmutzige Seiten dürfen jederzeit ersetzt und in die DB eingebracht werden
- DB kann unbestätigte Änderungen enthalten
- *UNDO*-Recovery ist erforderlich
- effektivere Puffernutzung bei langen TAs mit vielen Änderungen



## Pufferverwaltung (2)

- **Ausschreibestrategien (*EOT*-Behandlung)**

- **Force**

- Alle geänderte Seiten werden spätestens bei *EOT* (vor *COMMIT*) in die DB geschrieben
- keine *REDO*-Recovery erforderlich bei Systemfehler
- sehr hoher I/O-Aufwand, da Änderungen jeder TA einzeln geschrieben werden
- Vielzahl an Schreibvorgängen führt zu schlechteren Antwortzeiten, länger gehaltenen Sperren und damit zu mehr Sperrkonflikten
- Große DB-Puffer werden schlecht genutzt

- **No-Force**

- Änderungen können auch erst nach dem *COMMIT* in die DB geschrieben werden
- Beim *COMMIT* werden lediglich *REDO*-Informationen in die Log-Datei geschrieben
- *REDO*-Recovery erforderlich bei Systemfehler
- Änderungen auf einer Seite über mehrere TAs hinweg können gesammelt werden



## Pufferverwaltung (3)

- Kombination der Verdrängungs- und Ausschreibestrategien

	No-Steal	Steal
Force	kein <i>UNDO</i> – kein <i>REDO</i> (nicht für Update-in-Place)	<i>UNDO</i> – kein <i>REDO</i>
No-Force	kein <i>UNDO</i> – <i>REDO</i>	<i>UNDO</i> – <i>REDO</i>

- Bewertung **Steal / No-Force**
  - erfordert zwar *UNDO* als auch *REDO*, ist aber allgemeinste Lösung
  - beste Leistungsmerkmale im Normalbetrieb
- Bewertung **No-Steal / Force**
  - optimiert den Fehlerfall auf Kosten des Normalfalls (sehr teures *COMMIT*)
  - für *Update-in-Place* nicht durchführbar:
    - wegen *No-Steal* dürfen Änderungen erst nach *COMMIT* in die DB gelangen, was jedoch *Force* widerspricht (*No-Steal* → *No-Force*)
    - wegen *Force* müssten Änderungen vor dem *COMMIT* in der DB stehen, was bei *Update-in-Place* unterbrochen werden kann, *UNDO* wäre nötig (*Force* → *Steal*)



## WAL-Prinzip und COMMIT-Regel

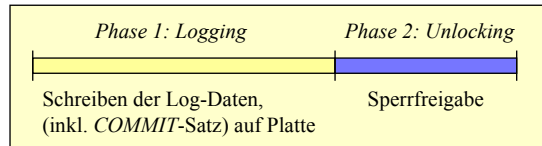
Fundamentale Regeln zum korrekten Wiederanlauf im Fehlerfall:

- **WAL-Prinzip (*Write-Ahead-Log*)**
  - *UNDO*-Information (z.B. *BFIM*) muss vor Änderung der DB im Protokoll stehen
  - Wichtig, um schmutzige Änderungen rückgängig machen zu können
  - Nur relevant für *Steal*
  - Wichtig bei direktem Einbringen
- **COMMIT-Regel (*Force-Log-at-Commit*)**
  - *REDO*-Information (z.B. *AFIM*) muss vor dem *COMMIT* im Protokoll stehen
  - Voraussetzung zur Durchführbarkeit der Crash-Recovery bei *No-Force*
  - Erforderlich für Geräte-Recovery (auch bei *Force*)
  - Gilt für direkte und indirekte Einbringstrategien gleichermaßen
- Bemerkung:  
Um die chronologische Reihenfolge im Ringpuffer zu wahren, werden alle Log-Einträge bis zum letzten notwendigen ausgeschrieben, d.h. es werden keine Log-Einträge übergangen



## COMMIT-Verarbeitung (1)

### • Standard Zwei-Phasen-Commit

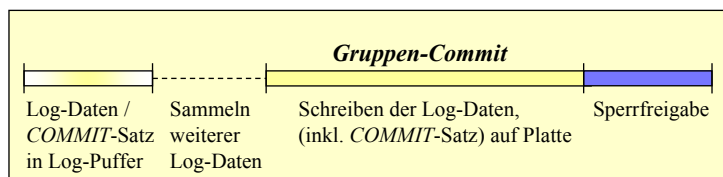


- Phase 1: Logging
  - Überprüfen der verzögerten Integritätsbedingungen
  - Logging der REDO-Informationen incl. COMMIT-Satz
- Phase 2: Unlocking
  - Freigabe der Sperren (Sichtbarmachen der Änderungen)
  - Bestätigung des COMMIT an das Anwendungsprogramm
- Problem: COMMIT-Regel verlangt Ausschreiben des Log-Puffers bei jedem COMMIT
  - Beeinträchtigung für kurze TAs, deren Log-Daten weniger als eine Seite umfassen
  - Durchsatz an TAs ist eingeschränkt



## COMMIT-Verarbeitung (2)

### • Gruppen-Commit

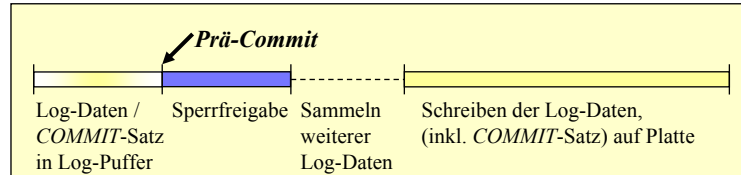


- Log-Daten mehrerer Transaktionen werden im Puffer gesammelt
- Log-Puffer wird auf Platte geschrieben, sobald Puffer gefüllt ist oder nach Timeout
- Vorteil: Reduktion der Plattenzugriffe und höhere Transaktionsraten möglich
- Nachteil: längere Sperrdauer führt zu längeren Antwortzeiten
- wird von zahlreichen DBS unterstützt



## COMMIT-Verarbeitung (3)

### • Prä-Commit



- Vermeidung der langen Sperrzeiten des Gruppen-Commit indem Sperren bereits freigegeben werden, wenn *COMMIT*-Satz im Log-Puffer steht
- Ist Prä-Commit zulässig?
  - *Normalfall*: ändernde TA kommt erfolgreich zu Ende, Änderungen sind gültig
  - *Fehlerfall*: Abbruch der TA nur noch durch Systemfehler möglich; bei Systemfehler werden auch die anderen laufenden TAs abgebrochen, "schmutziges Lesen" kann sich also nicht auf DB auswirken



## Inhalt

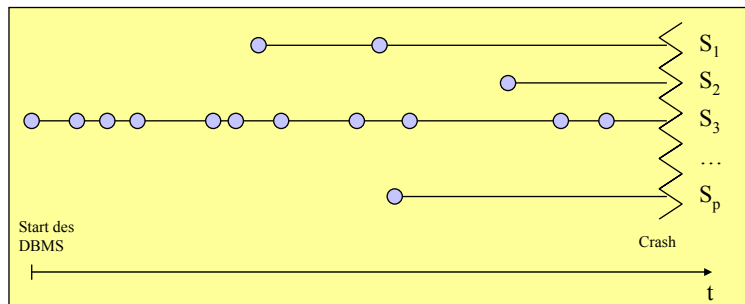
1. Recovery-Arten
2. Logging-Techniken
3. Abhängigkeiten zu anderen Systemkomponenten
4. Sicherungspunkte





## Sicherungspunkte (1)

- Maßnahme zur Begrenzung des *REDO*-Aufwands nach Systemfehlern
- Ohne Sicherungspunkte müssten potentiell alle Änderungen seit Start des DBMS wiederholt werden
- Besonders kritisch: Hot-Spot-Seiten, die (fast) nie aus dem Puffer verdrängt werden



## Sicherungspunkte (2)

- **Durchführung von Sicherungspunkten**
  - Spezielle Log-Einträge:  
BEGIN\_CHKPT  
Info über laufende TAs  
END\_CHKPT
  - *LSN* des letzten vollständig ausgeführten Sicherungspunktes wird in Restart-Datei geführt
- **Häufigkeit von Sicherungspunkten**
  - *zu selten*: hoher *REDO*-Aufwand
  - *zu oft*: hoher Overhead im Normalbetrieb
  - z.B. Sicherungspunkte nach bestimmter Anzahl von Log-Sätzen einfügen



## Direkte Sicherungspunkte

- **Charakterisierung**

- Alle geänderten Seiten werden in die persistente DB (Platte) geschrieben
- Zeitbedarf steigt mit dem zeitlichen Abstand der Sicherungspunkte
- Multi-Page-Access hilft, Schreibkopf-Positionierungen zu minimieren
- *REDO*-Recovery kann beim letzten vollständig ausgeführten Checkpoint beginnen

- **3 Arten**

- Transaktions-orientierte Sicherungspunkte (**TOC**)
- Transaktions-konsistente Sicherungspunkte (**TCC**)
- Aktions-konsistente Sicherungspunkte (**ACC**)



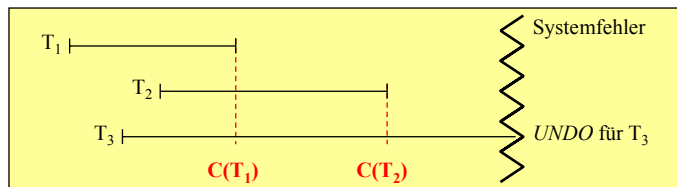
## Transaktions-orientierte Sicherungspunkte: **TOC** (1)

- **TOC** = *Force*, d.h. Ausschreiben aller Änderungen beim *COMMIT*
- Nicht alle Seiten im Puffer werden geschrieben, sondern nur Änderungen der jeweiligen TA
- Sicherungspunkt bezieht sich immer auf genau eine TA
- **UNDO-Recovery**  
Bei *Update-in-Place* ist *UNDO* nötig (*Force* → *Steal*), *UNDO* beginnt dann beim letzten Sicherungspunkt
- **REDO-Recovery** nicht nötig



## Transaktions-orientierte Sicherungspunkte: **TOC** (2)

- **Vorteile:**
  - keine *REDO* nötig
  - Implementierung ist einfach in Kombination mit Seitensperren
- **Nachteil:** (sehr) aufwändiger Normalbetrieb, insbesondere für Hot-Spot-Seiten
- **Beispiel:** Sicherungspunkte bei *COMMIT* von  $T_1$  und  $T_2$ , deshalb kein *REDO* nötig



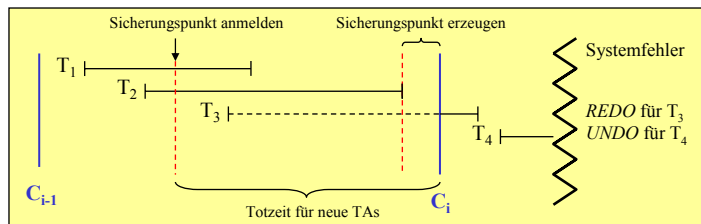
## Transaktions-konsistente Sicherungspunkte: **TCC** (1)

- DB wird in TA-konsistenten Zustand gebracht, d.h. keine schmutzigen Änderungen
- Während der Sicherung dürfen keine Änderungs-TAs aktiv sein
- Sicherungspunkt bezieht sich immer auf alle TAs
- *UNDO*- und *REDO-Recovery* sind durch letzten Sicherungspunkt begrenzt
- **Ablauf:**
  - Anmeldung des Sicherungspunktes
  - Warten, bis alle Änderungs-TAs abgeschlossen sind
  - Erzeugen des Sicherungspunktes
  - Verzögerung neuer Änderungs-TAs bis zum Abschluss der Sicherung



## Transaktions-konsistente Sicherungspunkte: **TCC** (2)

- **Vorteil:** *UNDO*- und *REDO*-Recovery beginnen beim letzten Sicherungspunkt (im Beispiel:  $C_i$ ), d.h. es sind nur TAs betroffen, die nach der letzten Sicherung gestartet wurden (hier:  $T_3, T_4$ )
- **Nachteil:** lange Wartezeiten (“Totzeiten”) im System
- **Beispiel:**



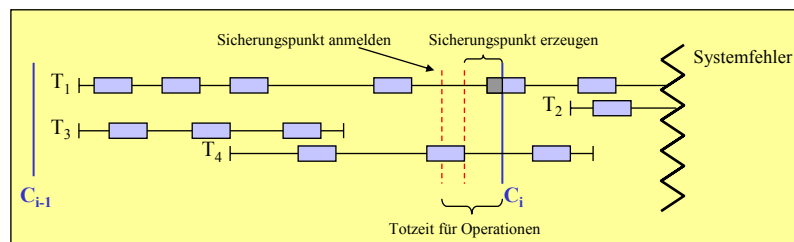
## Aktions-konsistente Sicherungspunkte: **ACC** (1)

- Blockierung nur auf Operationenebene, nicht mehr für ganze TAs
- Keine Änderungsoperationen während der Sicherung
- **UNDO-Recovery** beginnt bei  $MinLSN$  = kleinste  $LSN$  aller noch aktiven TAs des letzten Sicherungspunktes
- **REDO-Recovery** durch letzten Sicherungspunkt begrenzt
- **Ablauf:**
  - Anmelden des Sicherungspunktes
  - Beendigung aller laufenden Änderungsoperationen abwarten (im Beispiel:  $T_4$ )
  - Erzeugen des Sicherungspunktes
  - Verzögerung neuer Änderungsoperationen bis zum Abschluss der Sicherung (im Beispiel:  $T_1$ )



## Aktions-konsistente Sicherungspunkte: ACC (2)

- **Vorteil:** Totzeit des Systems für Änderungen deutlich reduziert
- **Nachteil:** Geringere Qualität der Sicherungspunkte
  - schmutzige Änderungen können in die Datenbank gelangen
  - zwar *REDO*-, nicht jedoch *UNDO*-Recovery durch letzten Sicherungspunkt begrenzt
- **Beispiel:**



## Indirekte Sicherungspunkte (1)

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



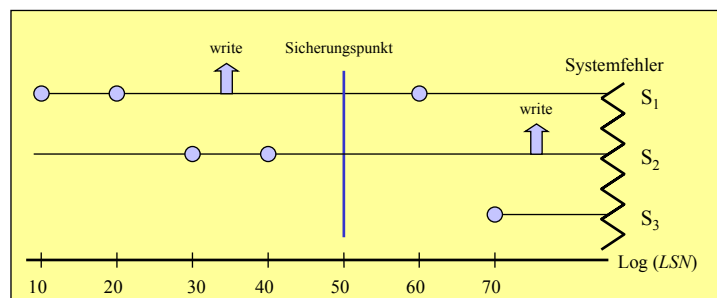
## Indirekte Sicherungspunkte (2)

- **Ausschreiben von DB-Änderungen**
  - außerhalb der Sicherungspunkte, asynchron zur laufenden TA-Verarbeitung
  - länger nicht mehr referenzierte Seiten werden vorausschauend ausgeschrieben
  - Sonderbehandlung für Hot-Spot-Seiten nötig, die praktisch nie ersetzt werden:
    - zwangsweises Ausschreiben bei bestimmtem Log-Umfang
    - Anlegen einer Kopie, um keine Verzögerung für neue Änderungen zu verursachen
- **UNDO-Recovery** beginnt bei  $MinLSN$
- **REDO-Recovery**
  - Startpunkt ist nicht mehr durch letzten Sicherungspunkt gegeben, auch weiter zurückliegende Änderungen müssen ggf. wiederholt werden
  - Zu jeder geänderten Seite wird  $StartLSN$  vermerkt ( $LSN$  der 1. Änderung seit Einlesen von Platte)
  - $REDO$  beginnt bei  $MinDirtyPageLSN = \min(StartLSN)$



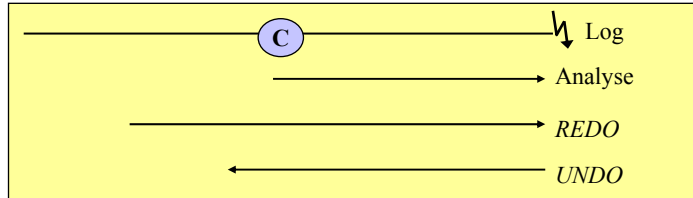
## Indirekte Sicherungspunkte (3)

- **Beispiel:**
  - beim Sicherungspunkt stehen  $S_1$  und  $S_2$  geändert im Puffer
  - älteste noch nicht ausgeschriebene Änderung ist auf Seite  $S_2$
  - $MinDirtyPageLSN$  hat also den Wert 30, dort muss  $REDO$ -Recovery beginnen





## Allgemeine Prozedur der Crash-Recovery (1)



### 1. Analyse-Phase

- Lies Log-Datei vom letzten Sicherungspunkt bis zum Ende
- Bestimmung von Gewinner- und Verlierer-TAs, sowie der Seiten, die von ihnen geändert wurden
  - *Gewinner*: TAs, für die ein *COMMIT*-Satz im Log vorliegt
  - *Verlierer*: TAs, für die ein *ROLLBACK*-Satz bzw. kein *COMMIT*-Satz vorliegt
- Ermittle alle weiteren Seiten, die nach dem Checkpoint geändert wurden



## Allgemeine Prozedur der Crash-Recovery (2)

### 2. REDO-Phase

- Vorwärtslesen der Log-Datei: Startpunkt ist abhängig vom Sicherungspunkttyp
- Aufgabe: Wiederholen der Änderungen, die noch nicht in der DB vorliegen
- zwei Ansätze:
  - vollständiges *REDO* (*redo all*): Alle Änderungen werden wiederholt
  - selektives *REDO*: Nur die Änderungen der Gewinner-TAs werden wiederholt



## Allgemeine Prozedur der Crash-Recovery (3)

### 3. *UNDO*-Phase

- Rückwärtslesen der Log-Datei bis zum *BOT*-Satz der ältesten Verlierer TA
- Aufgabe: Zurücksetzen der Verlierer-TAs
- Fertig wenn Beginn der ältesten TA erreicht ist, die bei letztem Checkpoint aktiv war
- abhängig von *REDO*-Vorgehen:
  - vollständiges *REDO*: nur zum Fehlerzeitpunkt noch laufende TAs zurücksetzen
  - selektives *REDO*: alle Verlierer-TAs zurücksetzen (beendete und unbeendete)

### 4. Abschluß der Recovery durch einen Sicherungspunkt