Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Datenbanksysteme II
Recovery
(Kapitel 17)

7.7.2008

Felix Naumann

Wdh: Fehlerklassifikation

1. Transaktionsfehler
   - Führt zu Transaktionsabbruch
   - Fehler in der Anwendung (division by zero)
   - **`abort`** Befehl
   - Abbruch durch DBMS (Deadlock)
2. Systemfehler
   - Absturz in DBMS, Betriebssystem, Hardware
   - Daten im Hauptspeicher werden zerstört
3. Medienfehler
   - Head-crash, Controller-Fehler, Katastrophe
   - Daten auf Disk werden zerstört

Überblick

- Fehlerarten
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Archivierung

Widerstandsfähigkeit (*Resilience*)

- Integrität der Datenbank trotz Fehler wiederherstellen
  - Systemfehler => Logging
  - Medienfehler => Archivierung
- Fehlerarten (etwas feiner)
  - Fehlerhaften Dateneingabe
  - Medienfehler
  - Katastrophe
  - Systemfehler

Fehlerhafte Dateneingabe

- Automatische Entdeckung im Allgemeinen nicht möglich
  - Wie doch?
- Nebenbedingungen
  - Schlüssel und Fremdschlüssel
  - Datentypen
  - CHECK constraints
    - Anzahl Ziffern in Telefonnummer
    - Fester Wertebereich
- Trigger
  - Z.B. zur Musterprüfung
- Nach erfolgter Eingabe: Datenreinigung

Medienfehler

- Einfache Fehler
  - Wenige Bit kaputt
  - Parity Bits (siehe VL „Physische Speicherstrukturen“)
- Schwerwiegende Fehler
  - Gesamte Festplatte unleserlich
  - Verschiedene RAID Stufen (siehe VL „Physische Speicherstrukturen“)
  - Archivierung (später in dieser VL)
    - Auf DVD oder Magnetband
    - Periodisch erzeugt
    - Aufbewahrung an anderem Ort
  - Verteilte Datenbanken
    - Parallele Datenhaltung an mehreren Sites

Katastrophe

- Explosion, Feuer, Flut, Vandalismus
- Alle Medien vollständig zerstört

- Abhilfe
  - RAID hilft nicht mehr
  - Archivierung
  - Verteilte Datenbanken

Systemfehler

- Transaktionen
  - Haben während des Ablaufs einen Zustand
    - Werte für Variablen
    - Aktuelles Code-Stück
  - Zustand im Hauptspeicher
  - Zustand geht bei Systemfehler verloren
- Stromausfall
  - Zustand gelöscht
- Softwarefehler
  - Zustand überschrieben

- Abhilfe
  - Logging

Wdh: Die Transaktion

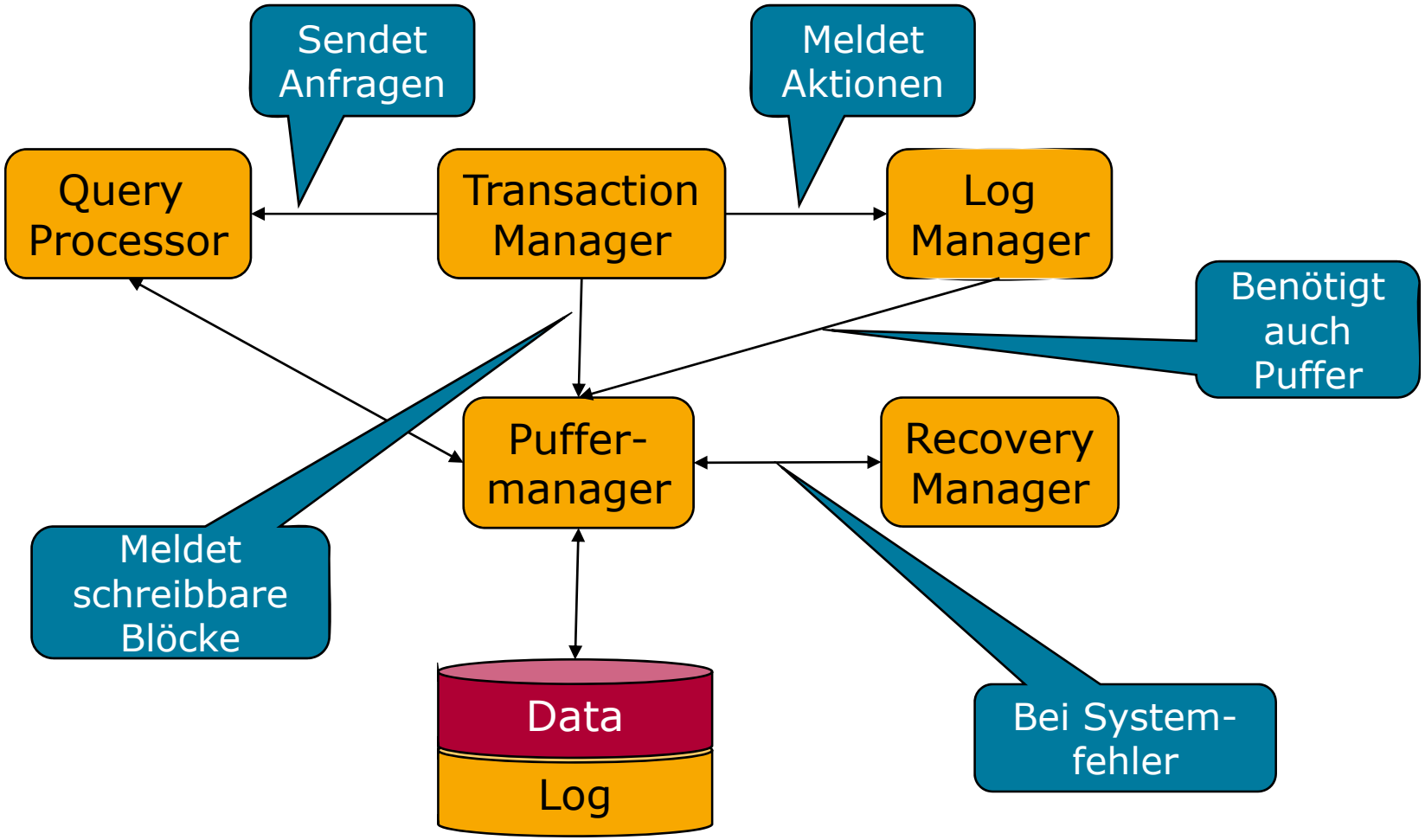
- Eine Transaktion ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen konsistenten (eventuell veränderten) Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss.
- Wdh.: Was war ACID?

- Aspekte:
  - Semantische Integrität: Korrekter (konsistenter) DB-Zustand nach Ende der Transaktion
  - Ablaufintegrität: Fehler durch „gleichzeitigen" Zugriff mehrerer Benutzer auf dieselben Daten vermeiden

Transaktionen

- Bisher aus Sicht des Entwicklers
  - Einzelne SQL Befehle sind Transaktionen
  - Embedded SQL: Folge von SQL Befehlen
    - COMMIT bzw. ROLLBACK am Ende
- Atomarität gilt.
- Nun aus Sicht des Transaktionsmanager
  - Verantwortlich für korrekten Ablauf der Transaktion
  - Gibt Meldungen an den Log Manager
  - Nebenläufigkeit / Scheduling
  - Muss Atomarität gewährleisten

Transaktions- und Log-Manager



Ausführung von Transaktionen

- Datenbank besteht aus „Elementen“
  - Definition von „Element“ je nach DBMS
    - Ganze Relationen
    - Blöcke
    - Tupel
- Datenbank hat Zustand: Wert für jedes Element
  - Konsistent bzw. inkonsistent
  - Konsistent = Alle expliziten und impliziten Nebenbedingungen sind erfüllt
    - Implizit: Trigger, Transaktionen, etc.
- Annahme der Korrektheit: Wenn eine Transaktion isoliert und ohne Systemfehler auf einer konsistenten Datenbank ausgeführt wird, ist die Datenbank nach der Transaktion wiederum konsistent
  - Das „C“ in ACID

Verlauf einer Transaktion

- Adressräume
  - Blocks auf Disk mit Datenbankelementen
  - Virtueller oder Hauptspeicher
    - Puffermanager (*buffer manager*)
  - Lokaler Adressraum der Transaktion
- Vorgehen einer Transaktion
  1. Anfordern eines Datenbankelements beim Puffermanager
  2. Puffermanager holt gegebenenfalls Element von Disk
  3. Element wird in lokalen Adressraum der Transaktion geholt
  4. Gegebenenfalls wird in dem Adressraum neues/verändertes Element erzeugt.
  5. Rückgabe des neuen Elements an Puffermanager
  6. Irgendwann: Puffermanager schreibt Element auf Disk
- Widerstandfähigkeit: Häufiges Schreiben auf Disk
- Effizienz: Seltenes Schreiben auf Disk

Operationen

- Bewegen von Elementen zwischen Adressräumen
- **`INPUT(X)`**
  - Kopiert **`X`** von Disk in Puffer
  - Ausgeführt von Puffermanager
- **`READ(X, t)`**
  - Kopiert **`X`** von Puffer in Transaktionsadressraum
  - **`t`** ist lokale Variable
  - Ausgeführt von Transaktion
- **`WRITE(X, t)`**
  - Kopiert Wert von **`t`** auf Element **`X`** im Puffer
  - Ausgeführt von Transaktion
- **`OUTPUT(X)`**
  - Kopiert Block mit **`X`** von Puffer auf Disk
  - Ausgeführt von Puffermanager
- Annahme: Ein Element passt immer in einen Block
  - Falls nicht zutreffend: Aufteilen von Elementen in einzelne Blöcke
  - Sorgfältig sein: Immer alle Blöcke schreiben

Operationen – Beispiel

- Zwei Datenbankelemente: A und B
  - Nebenbedingung: A = B
  - Realistischere Nebenbedingungen:
    - Verkaufte Flüge < 110% der Sitzplätze
    - Summe der Kontostände = Gesamtguthaben
- Transaktion T
  - A := A · 2
  - B := B · 2
- T ist in sich konsistent

Operationen – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t := t · 2	16	8		8	8
WRITE(A,t)	16	16		8	8
READ(B,t)	8	16	8	8	8
t := t · 2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

- Wo darf ein Systemfehler geschehen?
- Wo darf kein Systemfehler geschehen?

Überblick

- Fehlerarten
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Archivierung

Logging

- Ziel: Atomarität der Transaktionen
- Log = Folge von „log records“ (Log-Datensätze)
- Aktionen verschiedener Transaktionen können sich überlappen.
  - Deshalb: Nicht erst am Ende der Transaktion die gesammelten Aktionen loggen.
- Systemfehler: Log zur Rekonstruktion eines konsistenten Zustandes verwenden
- Medienfehler: Log und Archiv verwenden
- Allgemein
  - Einige Transaktionen müssen wiederholt werden (redo).
  - Einige Transaktionen müssen rückgängig gemacht werden (undo).
- Jetzt: Undo-logging
  - Falls nicht absolut sicher ist, dass das Ergebnis einer Transaktion vollständig auf Disk geschrieben wurde: Undo!
  - Nur undo, kein redo

Log Datensätze

- Logdatei: Append-only
- Log-Manager speichert jedes wichtige Ereignis.
- Logdatei ebenfalls in Blöcke organisiert
  - Zunächst im Hauptspeicher
  - Dann Schreiben auf Disk
  - Dies verkompliziert logging weiter!
- Verschiedene Log-Datensätze
  - <START T> - Transaktion T hat begonnen.
  - <COMMIT T> - Transaktion T ist beendet und wird nichts mehr ändern.
    - Allgemein: Änderungen sind noch nicht unbedingt auf Disk (je nach Puffermanager Strategie).
    - Undo-logging verlangt dies jedoch.
  - <ABORT T> - Transaktion T ist abgebrochen.
    - Transaction-Manager muss sicherstellen, dass T keinen Effekt auf Disk hat.
  - <T, X, v> - Update: Transaktion T hat X verändert; alter X-Wert ist v
    - Speziell für Undo-Logging: Neuer Wert nicht benötigt
    - Wird nach einem WRITE geschrieben (nicht erst nach einem OUTPUT)
      - Warum?

Regeln des Undo-Logging

- U1: Falls Transaktion T Element X verändert, muss <T, X, v> auf Disk geschrieben sein BEVOR neuer Wert von X auf Disk geschrieben wird.
  - Puffermanager muss aufpassen.
- U2: Falls eine Transaktion committed: <COMMIT T> darf erst ins Log geschrieben werden NACHDEM alle veränderten Elemente auf Disk geschrieben wurden.
  - Puffermanager muss aufpassen.
- Es ergibt sich folgende Abfolge:
  1. Log-Datensatz für veränderte Elemente schreiben
  2. Elemente selbst schreiben
  3. COMMIT Log-Datensatz schreiben
  - 1. und 2. separat für jedes Element!

Undo-Logging –Ursprünge

http://www.mlahanas.de/Greeks/Mythology/Labyrinth.html

Log-Manager

- Log-Manager: *Flush-log* Befehl weist Puffermanager an, alle Log-Blöcke auf Disk zu schreiben.
- Transaktionsmanager: OUTPUT Befehl weist Puffermanager an, Element auf Disk zu schreiben.

- Größe eines Update-Datensatzes im Log?
  - Insbesondere falls Element = Block: Update-Datensatz kann größer als Block werden
  - Aber: Komprimierte Darstellung: Nur Änderungen
    - Z.B. nur auf einem Attribut

Undo-Logging – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Nötig?

Nein, aber T kann lange als committed erscheinen, wird aber bei einem Recovery zurückgesetzt.

Parallelität von Transaktionen

- Viele Transaktionen schreiben gleichzeitig ins Log.
- FLUSH LOG wird immer wieder ausgeführt.
  - Auch durch fremde Transaktionen
  - Schlimm?
  - Nur <COMMIT T> darf nicht frühzeitig geschrieben werden.
    - Kann aber nicht passieren, da erst spät genug im Log
- Problem: Zwei Transaktionen fassen Tupel auf gleichem Block an.
  - Transaktion T1 führt OUTPUT durch bevor T2s Log-Datensätze geflushed werden
  - Lösung?
    - Block als Granularität von Sperrprotokollen

Recovery mittels Undo-Logging

- Problem: Systemfehler mitten während Transaktionen
  - Atomarität nicht erfüllt
  - Datenbank nicht in konsistentem Zustand
  - Recovery Manager muss helfen
- Zunächst: Naiver Ansatz – ganzes Log betrachten
- Später: Checkpointing – Nur Log nach einem Checkpoint betrachten

- Aufteilung aller Transaktionen
  - „Committed" falls <COMMIT T> vorhanden
  - „Uncommitted" sonst (<START T> ohne <COMMIT T>)
    - Undo nötig!
    - Verwendung der Update-Datensätze im Log
- Schreibe alten Wert v für Element X gemäß Update-Datensatz.
  - Egal was aktueller Wert ist

Recovery mittels Undo-Logging

- Probleme:
  - Mehrere uncommitted Transaktionen
  - Mehrere uncommitted Transaktionen haben X verändert.
- Lösung
  - Log-Datei von Ende nach Anfang durchgehen
    - Umgekehrt chronologisch
  - Jüngste Werte zuerst
- Beim Wandern zurück
  - Merke alle Transaktionen mit COMMIT oder ABORT
  - Bei Update-Datensatz <T, X, v>
    - Falls für T ein COMMIT oder ABORT vorhanden: Tue nichts
    - Falls nicht: Schreibe v auf X
- Am Ende
  1. <ABORT X> für alle uncommitted Transaktionen in Log schreiben
  2. FLUSH LOG

Recovery mittels Undo-Logging – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Bei Recovery:
- T wird als committed erkannt
- Keine Wiederherstellung nötig

Crash

Recovery mittels Undo-Logging – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Bei Recovery:
- Vielleicht ist <COMMIT T> Datensatz bereits zufällig auf Disk gelandet.
  - Wird als committed erkannt
  - Nichts tun
- Sonst: T wird als uncommitted erkannt
  - Wert 8 wird auf B geschrieben
  - Wert 8 wird auf A geschrieben
  - <ABORT T> wird in Log geschrieben
  - Log wird geflushed

Crash

Recovery mittels Undo-Logging – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(						<T, A, 8>
READ(B						
t := t ·						
WRITE(						<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Bei Recovery:
- T wird (sicher) als uncommitted erkannt
  - Wert 8 wird auf B geschrieben
  - Wert 8 wird auf A geschrieben
  - <ABORT T> wird in Log geschrieben
  - Log wird geflushed

Crash

Recovery mittels Undo-Logging – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t*2	16	16		8	8	
WRITE(A,t)	16	16	8	8	8	<T, A, 8>
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG	16	16	16	8	8	
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Bei Recovery:
- T wird als uncommitted erkannt
  - Wert 8 wird auf B geschrieben (auch wenn Änderung noch gar nicht auf Disk angekommen ist)
  - Wert 8 wird auf A geschrieben (auch wenn Änderung noch gar nicht auf Disk angekommen ist)
  - <ABORT T> wird in Log geschrieben
  - Log wird geflushed

Crash

Recovery mittels Undo-Logging – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8		
t := t*2	16	16		8		
WRITE(A,t)	16	16		8		<T, A, 8>
READ(B,t)	8	16	8	8	8	
t := t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Bei Recovery:
- T wird als uncommitted erkannt
  - Problem: Sind Log-Datensätze schon auf Disk angekommen?
    - Egal, da vorher auch nicht Datenänderung geschehen konnte
  - Wert 8 wird auf B geschrieben (auch wenn Änderung noch gar nicht auf Disk angekommen ist)
  - Wert 8 wird auf A geschrieben (auch wenn Änderung noch gar nicht auf Disk angekommen ist)
  - <ABORT T> wird in Log geschrieben
  - Log wird geflushed

Crash

Systemfehler während Recovery

- Was tun?

- Bei Systemfehler: Einfach nochmals von vorn anfangen
- Recovery-Schritte sind *idempotent*.
  - Wiederholte Anwendung entspricht einfacher Anwendung.

Checkpointing

- Problem: Recovery zwingt, das gesamte Log zu lesen
  - Idee 1: Sowie man ein COMMIT sieht, abbrechen
    - Aber: Wg. paralleler Transaktionen können einige TAs dennoch uncommitted sein.
  - Idee 2: Periodisch *Checkpoint* setzen
    1. Vorübergehend neue Transaktionen ablehnen
    2. Warten bis alle laufenden Transaktion committed oder aborted sind und entsprechender Log-Datensatz geschrieben wurde
    3. Flush-log
    4. Log-Datensatz **<CKPT>** schreiben
    5. Flush-log
    6. Neue Transaktionen wieder annehmen
  - Änderungen aller vorigen Transaktionen sind auf Disk geschrieben.
  - Recovery muss nur bis **<CKPT>** Log lesen

Checkpointing – Beispiel

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <T2, B, 10>
5. <T2, C, 15>
6. <T1, D, 20>
7. <COMMIT T1>
8. <COMMIT T2>
9. <CKPT>
10. <START T3>
11. <T3, E, 25>
12. <T3, F, 30>

Entscheidung für Checkpoint

Nicht-blockierendes Checkpointing

- Problem: Datenbank wird während des Checkpointing blockiert
  - Keine neuen Transaktionen werden angenommen.
  - Gerade laufende Transaktionen könnten noch viel Zeit benötigen.
- Idee: Checkpoint nur für bestimmte Transaktionen
  1. Schreibe Log-Datensatz **<START CKPT (T1, …, Tk)>**
     - Alle aktiven Transaktionen
  2. Flush-log
  3. Warte bis T1, …, Tk committed oder aborted sind
     - Erlaube aber neue Transaktionen!
  4. Schreibe Log-Datensatz **<END CKPT>**
  5. Flush-log

Nicht-blockierendes Checkpointing – Recovery

- Lesend von hinten erst <END CKPT> oder erst <START CKPT …>?
  - Zuerst <END CKPT>
    - Recovery nur bis zum nächsten <START CKPT>
  - Zuerst <START CKPT T1,…, Tk>
    - D.h. Systemfehler während Checkpointing
    - Zu diesem Zeitpunkt sind T1, …, Tk die einzigen aktiven Transaktionen.
    - Recovery weiter rückwärts, aber nur bis zum Start der frühesten Transaktionen aus T1, …, Tk
      - Schnell durch geeignete Pointerstruktur in der Log-Datei

Nicht-blockierendes Checkpointing – Beispiele

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <T2, B, 10>

Entscheidung für nicht-blockierenden Checkpoint

5. <START CKPT (T1, T2)>
6. <T2, C, 15>
7. <START T3>
8. <T1, D, 20>
9. <COMMIT T1>
10. <T3, E, 25>
11. <COMMIT T2>
12. <END CKPT>
13. <T3, F, 30>

CRASH

Recovery ab Zeile 5

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <T2, B, 10>

Entscheidung für nicht-blockierenden Checkpoint

5. <START CKPT (T1, T2)>
6. <T2, C, 15>
7. <START T3>
8. <T1, D, 20>
9. <COMMIT T1>
10. <T3, E, 25>

CRASH

Recovery ab Zeile 3

Überblick

- Fehlerarten
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Archivierung

Undo vs. Redo Logging

- Problem bei Undo-Logging: „Commit" erst nachdem auf Disk geschrieben wurde
  - Potenziell hohe I/O-Kosten
- Vollständige und unvollständige Transaktionen
  - Undo macht unvollständige Transaktionen rückgängig und ignoriert vollständige Transaktionen
  - Redo ignoriert unvollständige Transaktionen und wiederholt vollständige Transaktionen
- COMMIT im Log
  - Undo: Erst nachdem alle Werte auf Disk sind
  - Redo: BEVOR irgendein Werte auf Disk geschrieben wird
- Werte im Log
  - Undo: Alte Werte
  - Redo: Neue Werte

Redo-Logging Regeln

- Log-Datensatz <T, X, v>
  - Transaktion T hat neuen Wert V für Datenbankelement X geschrieben.
- Redo Regel (Auch *write-ahead logging rule*)
  - R1: Bevor ein Datenbankelement X auf Disk verändert werden kann, müssen alle zugehörigen Log-Datensätze auf Disk geschrieben sein.
    - Zugehörig: Update-Datensatz und COMMIT
- Ablauf
  1. Update Log-Datensätze auf Disk schreiben
  2. COMMIT Log-Datensatz auf Disk schreiben
  3. Veränderte Datenbankelement auf Disk schreiben

Redo-Logging – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Redo-Logging – Recovery

- Beobachtung
  - Falls kein COMMIT im Log, sind Elemente auf Disk unberührt
    - Müssen also nicht wiederhergestellt werden
  - => Unvollständige Transaktionen können ignoriert werden.
- Committed Transaktionen sind Problem
  - Unsicher, welcher Änderungen auf Disk gelangt sind
  - Aber: Log-Datensätze haben alle Informationen
- Vorgehen
  1. Identifiziere committed Transaktionen
  2. Lese Log-Daten von Anfang bis Ende (chronologisch)
     1. Für jeden Update-Datensatz **<T, X, v>**
        1. Falls T nicht committed: Ignorieren
        2. Falls T committed: Schreibe v als Element X
  3. Für jede uncommitted Transaktion schreibe **<ABORT T>** in Log
  4. Flush-log

Redo-Logging – Recovery Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Bei Recovery:
- T wird als committed erkannt
  - Wert 16 wird auf A geschrieben (gegebenenfalls redundant)
  - Wert 16 wird auf B geschrieben (gegebenenfalls redundant)

Crash

Redo-Logging – Recovery Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Bei Recovery:
- Falls <COMMIT T> (zufällig) auf Disk gelangte
  - Wie vorher
- Falls nicht
  - Wie nächste Folie

Crash

Redo-Logging – Recovery Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 16>
READ(B,t)	8	8		8	8	
t := t · 2	16	16		8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Bei Recovery:
- <COMMIT T> kann nicht auf Disk gelangt sein
- T ist also unvollständig (uncommitted)
  - Zunächst nichts tun
- <ABORT T> in Log auf Disk schreiben

Crash

Redo-Logging – Checkpointing

- Neues Problem im Gegensatz zu Undo-Logging:
  - Beschränkung auf aktive Transaktionen genügt nicht:
    - COMMIT im Log obwohl Ergebnis noch nicht auf Disk geschrieben wurde.
- Idee: Während des Checkpointing alle Datenbankelemente auf Disk schreiben, die von committed aber noch nicht beendeten Transaktionen verändert wurden.
  - Puffermanager muss *dirty* Puffer kennen.
  - Log-Manager muss wissen, welche Transaktion welchen Puffer verändert hat.
- Vorteil: Man muss nicht auf die Beendigung aktiver (also uncommitted) Transaktionen warten,
  - denn die dürfen zurzeit eh nichts schreiben.

Redo-Logging – Checkpointing

- Vorgehen für nicht-blockierendes Checkpointing
  1. Schreibe <START CKPT (T1, …, Tk)> in Log
     - T1, …, Tk sind alle aktiven Transaktionen
  2. Flush-log
  3. Schreibe auf Disk alle Elemente,
     1. die im Puffer verändert,
     2. committed,
     3. aber noch nicht auf Disk geschrieben wurden.
  4. Schreibe <END CKPT> in Log
  5. Flush-log

Redo-Logging – Checkpointing Beispiel

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

Wert von A muss vor Ende des Checkpoint auf Disk geschrieben werden.

Entscheidung für nicht-blockierenden Checkpoint

Redo-Logging – Checkpointing Recovery

- Wie beim Undo Logging ist entscheidend, ob letzter Checkpoint-Datensatz in Log ein START oder ein END ist:
- <END CKPT>
  - Alle Transaktionen, die vor dem zugehörigen <START CKPT (T1, …, Tk)> committed sind, sind auf Disk.
  - T1,…, Tk und alle nach dem START begonnenen Transaktionen sind unsicher
    - Selbst bei COMMIT
  - Es genügt Betrachtung ab dem frühesten <START Ti>
    - Rückwärtsverlinkung im Log hilft
- <START CKPT (T1, …, Ti)>
  - D.h. Systemfehler während eines Checkpoint
  - Sogar committed Transaktionen vor diesem Punkt sind unsicher.
  - Rückwärtssuche zum nächsten <END CKPT> und dann weiter rückwärts zum zugehörigen <START CKPT (S1, …, Sj>
    - Warum nicht direkt zum früheren <START CKPT (…)>?
  - Dann Redo aller Transaktionen, die nach diesem START committed wurden und Redo der Si.

Könnte selbst wieder unbeendet sein.

Redo-Logging – Checkpointing Recovery Beispiel

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

CRASH

- Rückwärtssuche findet **<END CKPT>** bei 10.
- Redo aller TAs, die nach zugehörigen **<START CKPT (T2)>** (Zeile 6.) starteten (also T3)
- Redo aller TAs der Liste in 6. (also T2)
- T2 und T3 wurden committed
  - Also Redo für beide
- Suche Rückwärts bis **<START T2>** (3.)
  - Redo für drei Updates

Redo-Logging – Checkpointing Recovery Beispiel

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>
10. <END CKPT>
11. <COMMIT T2>

CRASH

12. <COMMIT T3>

- Wie vorher.
- Aber T3 ist nicht committed.
  - Entsprechend kein Redo für T3
  - **<ABORT T3>** in Log ergänzen

Redo-Logging – Checkpointing Recovery Beispiel

1. <START T1>
2. <T1, A, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 10>
6. <START CKPT (T2)>
7. <T2, C, 15>
8. <START T3>
9. <T3, D, 20>

CRASH

10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

- Suche nach vorletztem **<START CKPT (T1, …, Tk)>**
  - Hier nicht vorhanden
  - Also von Beginn des Logs arbeiten
- T1 ist einzige committed Transaktion
- Redo für T1
- **<ABORT T2>** in Log
- **<ABORT T3>** in Log

Überblick

- Fehlerarten
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Archivierung

Best of both Worlds

- Nachteil Undo: Daten müssen sofort nach Ende der Transaktion geschrieben werden => Mehr I/Os
- Nachteil Redo: Alle veränderten Blocks müssen im Puffer verbleiben bis COMMIT und Log-Datensätze auf Disk sind => Hoher Speicherbedarf
- Widersprüchliche Vorgaben (beide Ansätze)
  - A und B liegen auf einem Block.
  - A wurde von T1 verändert; T1 ist committed => Block muss auf Disk geschrieben werden.
  - B wurde von T2 verändert; T2 noch nicht committed => Block darf nicht auf Disk geschrieben werden.
- Undo/Redo Logging ist flexibler
  - Aber mehr Informationen in Log-Datensätzen nötig

Undo/Redo Regel

- Log-Datensatz: <T, X, v, w>
  - Alter Wert (v) und neuer Wert (w)
- Regel UR1:
  - Update-Datensatz <T, X, v, w> muss auf Disk geschrieben sein BEVOR das von T veränderte X auf Disk geschrieben wird.
- Diese Bedingung wird von Undo und Redo gemäß voriger Regeln ebenfalls verlangt.
  - Reminder U1: Falls Transaktion T Element X verändert, muss <T, X, v> auf Disk geschrieben sein BEVOR neuer Wert von X auf Disk geschrieben wird.
  - Reminder R1: Bevor ein Datenbankelement X auf Disk verändert werden kann, müssen alle zugehörigen Log-Datensätze auf Disk geschrieben sein.
- Was fehlt im Gegensatz zu Undo oder Redo?
  - <COMMIT T> darf VOR oder NACH Änderungen der Elemente auf Disk geschrieben werden.
    - Also keinerlei Einschränkung

Undo/Redo Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Mittendrin!

Hätte auch vorher oder nachher erscheinen können.

Undo/Redo Recovery

- Log erlaubt Redo und Undo durch Speicherung von neuem und altem Wert.
  - Redo für alle committed Transaktionen in chronologischer Reihenfolge
  - Undo aller uncommitted Transaktionen in umgekehrt-chronologischer Reihenfolge

- Beides ist wichtig: Erlaubte Extremfälle
  - Committed Transaktion mit keiner Änderung auf Disk
  - Uncommitted Transaktion mit allen Änderungen auf Disk

Undo/Redo Recovery – Beispiel

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

- <COMMIT T> eventl. schon auf Disk
- T scheint also committed
  - Aber B ist eventl. noch nicht auf Disk
- Bei Recovery werden beide Werte „16“ geschrieben.

Crash

Undo/Redo Recovery – Beispiel

- <COMMIT T> kann nicht auf Disk gelangt sein
- T ist also unvollständig (uncommitted)
- Recovery stellt beide Wert „8“ her, ob nötig oder nicht
- <ABORT T> auf Disk schreiben

Aktion	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t := t · 2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
READ(B,t)	8	16	8	8	8	
t := t · 2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Crash

Undo/Redo Recovery – Checkpointing

Einfacheres Vorgehen

1. Schreibe <START CKPT (T1, …, Tk)> in Log
   - Ti sind alle aktiven Transaktionen
2. Flush-log
3. Schreibe auf Disk alle zurzeit *dirty* Puffer
   - Also solche, die mindestens ein verändertes Element enthalten
   - Anmerkung: Bei Redo nur die Puffer mit committed Transaktionen
4. Schreibe <END CKPT> in Log
5. Flush-log

- Problem „Abort" und Dirty Data
  - Wichtige Annahme: Daten werden erst geschrieben, wenn eine Transaktion sicher ist, kein ABORT auszuführen.
    - Sowohl Hauptspeicher als auch Disk
  - Dirty Data trotz Serialisierbarkeit

Undo/Redo Recovery – Checkpointing Beispiel

1. <START T1>
2. <T1, A, 4, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 9, 10>
6. <START CKPT (T2)>
7. <T2, C, 14, 15>
8. <START T3>
9. <T3, D, 19, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

- Bei CKPT ist nur T2 aktiv
- <T2, B, 9, 10> eventuell schon auf Disk
  - Nicht möglich bei Redo-Logging
- Wird aber sowieso auf Disk geflushed
  - Wie alle dirty Blocks
  - Z.B. auch <T1, A, 4, 5> falls noch nicht geschehen.

Undo/Redo Recovery – Checkpointing Beispiel

1. <START T1>
2. <T1, A, 4, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 9, 10>
6. <START CKPT (T2)>
7. <T2, C, 14, 15>
8. <START T3>
9. <T3, D, 19, 20>
10. <END CKPT>
11. <COMMIT T2>
12. <COMMIT T3>

CRASH

- Findet <END CKPT>
- Und zugehöriges <START CKPT…>
- T1 irrelevant – ist committed vor dem <START CKPT…>
- Redo aller committed TAs der Liste und die nach dem <START CKPT…> starteten
  - Also T2 und T3
- Bei Redo nur bis zum <START CKPT…>
  - <T2, B, 9, 10> wurde ja schon auf Disk geschrieben

Undo/Redo Recovery – Checkpointing Beispiel

1. <START T1>
2. <T1, A, 4, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 9, 10>
6. <START CKPT (T2)>
7. <T2, C, 14, 15>
8. <START T3>
9. <T3, D, 19, 20>
10. <END CKPT>
11. <COMMIT T2>

CRASH

12. <COMMIT T3>

- T2 ist committed, T3 ist aktiv
- Redo für T2
  - Nur bis <START CKPT…>
- Undo für T3
  - Im Gegensatz zum Redo-logging: Dort kein Redo nötig
  - Hier kann neuer Wert schon auf Disk sein.

Undo/Redo Recovery – Checkpointing Beispiel

1. <START T1>
2. <T1, A, 4, 5>
3. <START T2>
4. <COMMIT T1>
5. <T2, B, 9, 10>
6. <START CKPT (T2)>
7. <T2, C, 14, 15>
8. <START T3>
9. <T3, D, 19, 20>
10. <END CKPT>

CRASH

11. <COMMIT T2>
12. <COMMIT T3>

- Undo für T2 und T3
- Da T2 aktiv zum Start des Checkpoints müssen auch Aktionen vor dem Checkpoint rückgängig gemacht werden.
  - Also auch <T2, B, 9, 10>

Überblick

- Fehlerarten
- Undo Logging
- Redo Logging
- Undo/Redo Logging
- Archivierung

Motivation

- Logging hilft bei Systemfehlern.
- Logging für Medienfehler?
  - OK, falls
    - Log auf getrennter Disk gespeichert wird,
    - und das Log vor einem Checkpoint nie verworfen wird
    - und redo bzw. undo/redo logging betrieben wird
      - Alte Werte helfen nicht bei der Wiederherstellung
  - Unrealistisch!
- Deshalb: Archivierung

Das Archiv

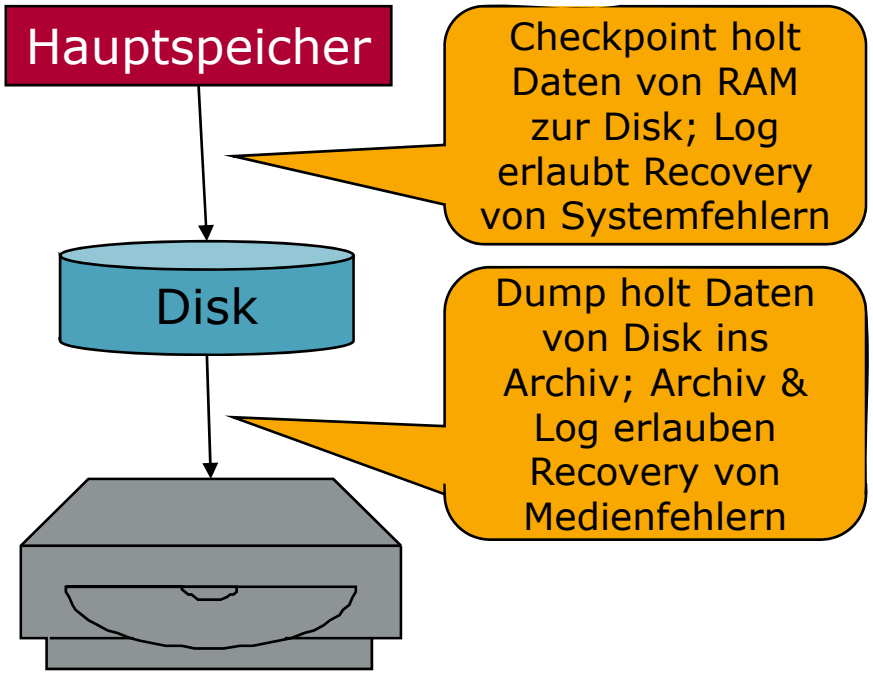
- Eine physische Kopie der Datenbank
  - Getrennt von der eigentlichen Datenbank
- Einfache Lösung
  - Hin und wieder Datenbank anhalten
  - Backup Kopie auf DVD oder Magnetband ziehen
    - An anderen Ort bringen
  - Wiederherstellung möglich für genau jenen Zeitpunkt
  - Anschließend das Log verwenden, um ab diesem Zeitpunkt die Datenbank weiterherzustellen.
    - Falls Log den Medienfehler überstanden hat
    - Z.B. durch schnelle Kopie des Logs zum selben Ort wie Archiv

Dumps / Backups

- Problem: Bei großen Datenbanken dauert Kopiervorgang zu lang
- Lösung: Zweistufiges Archivieren
  - *Full dump*: Kopie der gesamten Datenbank
  - *Incremental dump*: Nur veränderte Datenbankelemente kopieren
    - Veränderungen seit letztem full oder incremental dump
    - I.d.R. viel kleiner als full dump => Schneller durchzuführen!
- Noch feiner: *Level i dump*
  - Level-0 dump = full dump
  - Level-*i* dump kopiert alles Veränderte seit letztem level-*j* dump ($j \leq i$)
- Recovery nach Medienfehler
  - Kopiere full dump in Datenbank
  - Führe incremental dumps in chronologischer Reihenfolge nach.

Nicht-blockierendes Archivieren

- Problem: Selbst incremental dumps dauern mitunter zu lang.
  - Minuten bis Stunden
- Idee: Analog zu nicht-blockierendem Checkpointing
  - Nicht-blockierendes Archivieren kopiert Daten in fester Reihenfolge ins Archiv.
  - Während des dumps können Transaktionen die Datenbank weiter verändern.
  - Log kann verwendet werden, um diese Transaktionen bei der Recovery zu berücksichtigen.



Nicht-blockierendes Archivieren – Beispiel

- Datenbankelemente A, B, C, D
  - Mit initialen Werten 1, 2, 3, 4
  - Transaktionen ändern die Werte während des Dumps.
- Werte Nach Durchführung des Dumps?
  - Auf Disk: 5, 7, 6, 4
  - Im Archiv: 1, 2, 6, 4

Disk	Archiv
	Copy A
A := 5	
	Copy B
C := 6	
	Copy C
B := 7	
	Copy D

Nicht-blockierendes Archivieren

- Einzelschritte zum Archivieren
  - Annahme: Redo bzw. Undo/Redo logging
1. <START DUMP> in Log
2. Führe geeigneten Checkpoint durch
   - Je nach Logging Methode
3. Führe full bzw. incremental dump durch
   - Sicherstellen, dass die Kopie das andere Medium tatsächlich erreicht
4. Sicherstellen, dass Log mindestens bis zum Checkpoint ebenfalls archiviert ist
5. <END DUMP> in Log

- Anschließend kann Teil des Log verworfen werden. Welcher?
  - Log bis hin zum Checkpoint VOR dem Checkpoint des dumps.

Nicht-blockierendes Archivieren – Beispiel

- Zwei Transaktionen
  - T1 schreibt A und B
  - T2 schreibt C
  - Parallel dazu: Archivierung
- Log
  - <START DUMP>
  - <START CKPT (T1, T2)>
  - <T1, A, 1, 5>
  - <T2, C, 3, 6>
  - <COMMIT T2>
  - <T1, B, 2, 7>
  - <END CKPT>
  - dump endet…
  - <END DUMP>

Disk	Archiv
	Copy A
A := 5	
	Copy B
C := 6	
	Copy C
B := 7	
	Copy D

Recovery aus Archiv

- Schritte
  1. Rekonstruiere Datenbank aus Archiv
     1. Kopiere jüngsten *full dump* auf Disk
     2. Falls es spätere *incremental dumps* gibt: Führe entsprechende Änderungen auf Datenbank durch.
  2. Verändere Datenbank entsprechend des „überlebenden" Log-Teils.
     - ◇ Gleiche Methode wie gehabt

Recovery aus Archiv – Beispiel

- Gerettetes Log
  - <START DUMP>
  - <START CKPT (T1, T2)>
  - <T1, A, 1, 5>
  - <T2, C, 3, 6>
  - <COMMIT T2>
  - <T1, B, 2, 7>
  - <END CKPT>
  - dump endet…
  - <END DUMP>
- Insbesondere kein <COMMIT T1>
- Reminder: Nach Durchführung des Dumps
  - Werte auf Disk: 5, 7, 6, 4
  - Werte im Archiv: 1, 2, 6, 4
- Zunächst: Kopie des Archivs auf Disk
  - 1, 2, 6, 4
- T2 ist committed
  - Redo <T2, C, 3, 6>
  - Wert 6 steht schon in Datenbank – kann aber Zufall sein.
- T1 ist uncommitted
  - Undo <T1, B, 2, 7> und <T1, A, 1, 5>
  - Werte sind wiederum in Datenbank – kann wieder Zufall sein.

Zusammenfassung - Recovery

- Transaktionsmanagement
  - Resilience durch logging
  - Korrektheit durch scheduling
- Datenbankelemente
  - Einheit für Logging, Scheduling und Locking
  - Meist Disk Blöcke
  - Aber auch Relationen oder Tupel
- Logging
  - Datensatz für jede wichtige Aktion
  - Log muss auf Disk wenn geloggte Werte auf Disk wandern
    - Davor (undo) bzw. danach (redo)
- Recovery
  - Verwendung des Logs zum Wiederherstellen eines konsistenten Zustandes nach Systemfehler

- Undo Logging
  - Speichert alte Werte
  1. Log-Datensatz auf Disk
  2. Neuer Wert auf Disk
  3. Commit-Log auf Disk
  - Recovery durch Wiederherstellen alter Werte für uncommitted TAs
- Redo Logging
  - Speichert neue Werte
  1. Log-Datensatz auf Disk
  2. Commit-log auf Disk
  3. Neuer Wert auf Disk
  - Recovery durch Schreiben neuer Werte aller committed TAs
- Undo/Redo Logging
  - Speichert alte und neue Werte
  1. Log-Datensatz auf Disk
  2. Neuer Wert auf Disk
  - Commit-log egal wann

Zusammenfassung - Recovery

- Checkpointing
  - Erspart Betrachtung des gesamten Logs
  - Alte Log-Teile können gelöscht werden
- Nicht-blockierendes Checkpointing
  - Checkpointing unter Zulassung neuer TAs
  - Nachteil: Für einige TAs muss bei Recovery jenseits des Checkpoint geschaut werden.
- Archivierung
  - Schutz gegen Medienfehler
  - Speicherung von Datenbank and physisch entferntem (sicheren) Ort
- Inkrementelle Backups/Dumps
  - Nur Änderungen werden kopiert
  - Erspart Archivierung gesamter Datenbank
  - Nur Änderungen seit letztem (inkrementellen) dump
- Nicht-blockierendes Archivieren
  - Backup während Datenbank geändert wird
  - Log-Datensätze des Archivierungsvorgangs
  - Zusätzlicher Checkpoint
- Recovery aus Archiv
  1. Beginn mit full dump
  2. Nachführen der inkrementellen dumps
  3. Nachführen mittels archiviertem Log