

## 6. Grundlagen des Transaktionskonzepts

Stefan DeBloch

## Überblick

Atomarität von  
DB-Operationen  
und Trans-  
Aktionen?

Erhaltung der DB-  
Konsistenz

Anomalien im  
Mehrbenutzer-  
betrieb

Synchronisation  
von Transaktionen

Theorie der  
Serialisierbarkeit

Zwei-Phasen-  
Sperrprotokolle  
(2PL)

Logging und  
Recovery

Zwei-Phasen-  
Commit-  
Protokoll (2PC)

- **Wie erzielt man Atomarität von DB-Operationen und Transaktionen?**
  - Atomare Aktionen im Schichtenmodell
  - Schlüsselrolle von Synchronisation sowie Logging und Recovery
- **Erhaltung der DB-Konsistenz**
- **Anomalien im Mehrbenutzerbetrieb**
  - Verlorengegangene Änderungen
  - Inkonsistente Analyse, Phantom-Problem usw.
- **Synchronisation von Transaktionen**
  - Ablaufpläne, Modellannahmen
  - Korrektheitskriterium, Konsistenzerhaltende Ablaufpläne
- **Theorie der Serialisierbarkeit**
  - Äquivalenz von Historien, Serialisierbarkeitstheorem
  - Klassen von Historien
- **Zwei-Phasen-Sperrprotokolle (2PL)**
- **Logging und Recovery**
- **Zwei-Phasen-Commit-Protokoll (2PC)**

# What can go wrong, will go wrong ...

- **Transaktionskonzept**
  - führt ein neues Verarbeitungsparadigma ein
  - ist Voraussetzung für die Abwicklung betrieblicher Anwendungen (*mission-critical applications*)
  - erlaubt „**Vertragsrecht**“ in rechnergestützten IS zu implementieren
- **ACID-Transaktionen zur Gewährleistung weit reichender Zusicherungen zur Qualität der Daten, die gefährdet sind durch**
  - fehlerhafte Programme und Daten im Normalbetrieb
  - inkorrekte Synchronisation von Operationen im Mehrbenutzerbetrieb
  - vielfältige Fehler im DBS und seiner Umgebung
    - ➔ **Logging und Recovery bietet Schutz vor erwarteten Fehlern!**<sup>2</sup>
- **Entwicklungsziele**

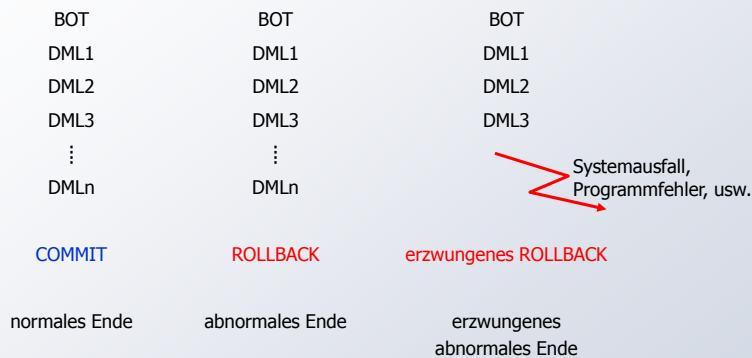
Build a system used by millions of people that is always available – out less than 1 second per 100 years = 8 9's of availability! (J. Gray: 1998 Turing Lecture)

  - Verfügbarkeit heute (optimistisch):<sup>3</sup>
    - für Web-Sites: 99%
    - für gut administrierte Systeme: 99,99%
    - höchstens: 99,999%
  - Künftige Verfügbarkeit
    - da fehlen noch 3 9-er
    - bis wann zu erreichen???

2. In petabyte systems, disk failures will daily occur (Q. Xin et al.)

3. Despite marketing campaigns promising 99,999% availability, well-managed servers today achieve 99,9% to 99%, or 8 to 80 hours downtime per year (Armando Fox)

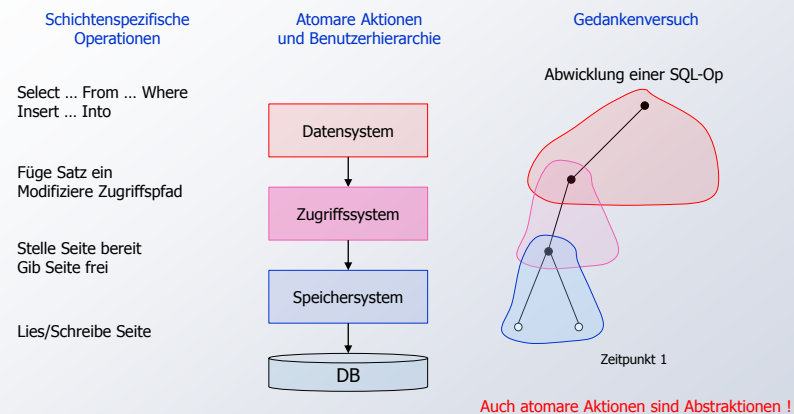
# Mögliche Ausgänge einer Transaktion



## Transaktionen als dynamische Kontrollstruktur

- **Atomicity**  
Atomarität ist keine natürliche Eigenschaft von Rechnern
  - **Consistency**  
Konsistenz und semantische Integrität der DB ist durch fehlerhafte Daten und Operationen eines Programms gefährdet.
  - **Isolation**  
Isolierte Ausführung bedeutet „logischen Einbenutzerbetrieb“
  - **Durability**  
Dauerhaftigkeit heißt, dass die Daten und Änderungen erfolgreicher Transaktionen jeden Fehlerfall „überleben“ müssen
- ➔ ACID-Transaktionen befreien den Anwendungsprogrammierer von den Aspekten der Ablaufumgebung des Programms und von möglichen Fehlern!
- Wie setzt man diese Forderungen systemtechnisch um?
    - hier nur Einführung von Begriffen
    - vertiefende Betrachtung und Diskussion von Realisierungskonzepten in nachfolgenden Vorlesungen (DBAW)

## Bausteine für Transaktionen – Atomare Aktionen



## Bausteine für Transaktionen – Atomare Aktionen (2)

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

Schichtenspezifische Operationen

Gedankenversuch

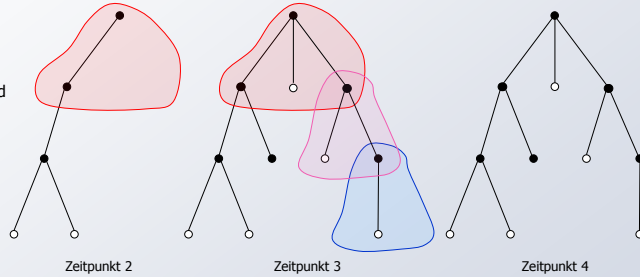
Abwicklung einer SQL-Op

Select ... From ... Where  
Insert ... Into

Füge Satz ein  
Modifiziere Zugriffspfad

Stelle Seite bereit  
Gib Seite frei

Lies/Schreibe Seite



Selbst wenn AA atomar implementiert wäre, Hierarchie von AA wäre es nicht!

## Schutzbedürfnis einer flachen Transaktion und Zusicherungen an den Programmierer

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

Schichtenspezifische Operationen

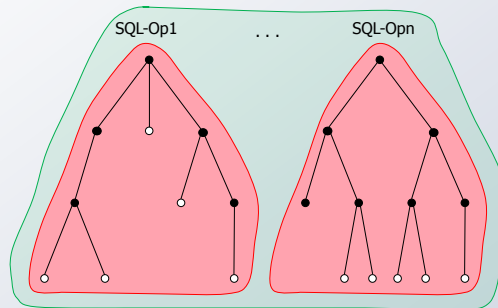
Schutzmaßnahmen im Ausführungspfad des DBS funktionieren nicht!

Select ... From ... Where  
Insert ... Into

Füge Satz ein  
Modifiziere Zugriffspfad

Stelle Seite bereit  
Gib Seite frei

Lies/Schreibe Seite



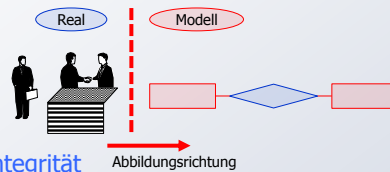
SQL garantiert Anweisungsatomarität und natürlich Transaktionsatomarität!

Realisierung verlangt vor allem

- Synchronisation im Mehrbenutzerbetrieb (concurrency transparency)
- Logging und Recovery (failure transparency)

## Erhaltung der DB-Konsistenz

### Abbildung der Miniwelt



### Erhaltung der semantischen Datenintegrität

- Beschreibung der „Richtigkeit“ von Daten durch Prädikate<sup>4</sup> und Regeln, bereits bekannt:
  - modellinhärente Bedingungen (relationale Invarianten)
  - anwendungsspezifische Bedingungen (Check, Unique, Not Null, ...)
- aktive Maßnahmen des DBS erwünscht (Trigger, ECA-Regeln)
- „Qualitätskontrollen“ bei Änderungsoperationen

### Ziel

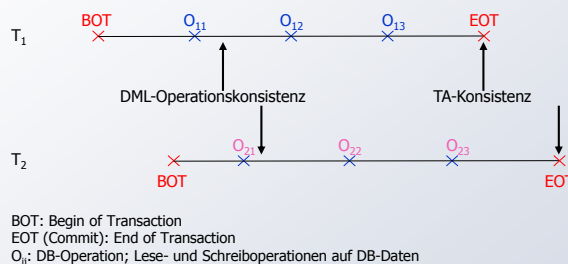
- Nur DB-Änderungen zulassen, die allen definierten *Constraints* entsprechen (offensichtlich „falsche“ Änderungen zurückweisen!)
  - Möglichst hohe Übereinstimmung von DB-Inhalt und Miniwelt (Datenqualität)
- ➔ *Integritätsbedingungen der Miniwelt sind explizit bekannt zu machen, um automatische Überwachung zu ermöglichen.*

4. „Ohne Ziel ist jeder Schuss ein Treffer“ oder „ohne Spezifikation von Constraints ist jeder Zustand richtig“.

## Erhaltung der DB-Konsistenz (2)

### Konsistenz der Transaktionsverarbeitung

- Bei COMMIT müssen alle Constraints erfüllt sein
- Zentrale Spezifikation/Überwachung im DBS: „*system enforced integrity*“



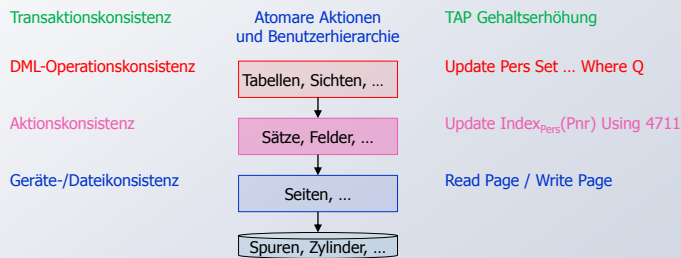
➔ C von ACID sichert dem Programmierer zu, dass vor BOT und nach EOT der DB-Zustand alle Constraints des DB-Schemas erfüllt!

## Erhaltung der DB-Konsistenz (3)

### Verfeinerung des Konsistenzbegriffes

- Transaktionsatomarität impliziert **Transaktionskonsistenz**: nur Änderungen erfolgreicher Transaktionen sind in der DB enthalten
- Anweisungsatomarität impliziert **DML-Operationskonsistenz**<sup>5</sup>: DML-Operation hält schichtenspezifische Konsistenz des Datensystems ein
- DML-Operationen setzen sich aus Aktionen zusammen: **Aktionskonsistenz** und **Geräte-/Dateikonsistenz** sind wiederum Voraussetzung, dass DML- Operationen und Aktionen überhaupt auf den Daten abgewickelt werden können

Systemhierarchie + DB-Konsistenz



5. „Golden Rule“ nach C. J. Date: No update operation must ever be allowed to leave any relation or view (relvar) in a state that violates its own predicate. Likewise no update transaction must ever be allowed to leave the database in a state that violates its own predicate.

## Erhaltung der DB-Konsistenz (5)

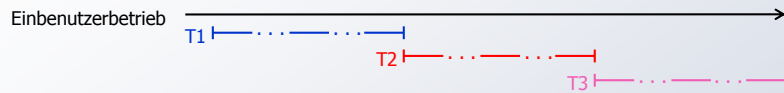
### Welche Konsistenzart garantiert jede Schicht nach erfolgreichem Abschluss einer schichtenspezifischen Operation?

- Speichersystem → Geräte-/Dateikonsistenz (einzelne Seite)  
Jede Seite muss physisch unversehrt, d. h. lesbar oder schreibbar sein
- Zugriffssystem → Aktionskonsistenz (mehrere Seiten)  
Sätze und Zugriffspfade müssen für Aktionen „in sich konsistent“ sein, d. h. beispielsweise: „Alle Zeiger müssen stimmen!“
- Datensystem → DML-Operationskonsistenz (oft viele Seiten)
- Datenbank → Transaktionskonsistenz  
Alle Constraints des DB-Schemas müssen erfüllt sein!

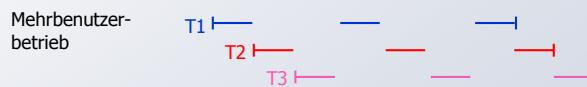
➔ **Konsistenz einer Schicht setzt schichtenspezifische Konsistenz aller darunter liegenden Schichten voraus!**

## Warum Mehrbenutzerbetrieb?

### Ausführung von Transaktionen



- CPU-Nutzung während TA-Unterbrechungen
  - E/A
  - Denkzeiten bei Mehrschritt-TA
  - Kommunikationsvorgänge in verteilten Systemen
- bei langen TA zu große Wartezeiten für andere TA (Scheduling-Fairness)



## Anomalien im unkontrollierten Mehrbenutzerbetrieb

1. Abhängigkeit von nicht freigegebenen Änderungen (*dirty read, dirty overwrite*)
2. Verlorengegangene Änderung (*lost update*)
3. Inkonsistente Analyse (*non-repeatable read*)
4. Phantom-Problem
5. Integritätsverletzung durch Mehrbenutzer-Anomalie
6. Instabilität von Cursor-Positionen

➔ nur durch Änderungs-TA verursacht

## Abhängigkeit von nicht freigegebenen Änderungen

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

T1	T2
read (A); A := A + 100 write (A);	read (A); read (B); B := B + A; write (B); commit;
abort;	

- Geänderte, aber noch nicht freigegebene Daten werden als „schmutzig“ bezeichnet (dirty data), da die TA ihre Änderungen bis Commit (einseitig) zurücknehmen kann
- ◊ Schmutzige Daten dürfen von anderen TAs nicht in „kritischen“ Operationen benutzt werden

## Verlorengegangene Änderung (Lost Update)

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

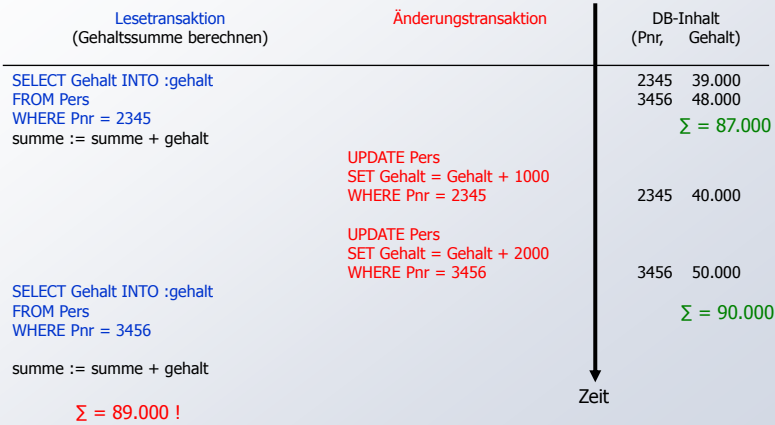
T1	T2	A in DB
read (A); 10		10
	read (A); 10	10
A := A - 2; 8 write (A)		10
		8
	A := A - 1; 9 write (A);	8
		9!

- ➔ Verlorengegangene Änderungen sind auszuschließen !



## Inkonsistente Analyse (Non-repeatable Read)

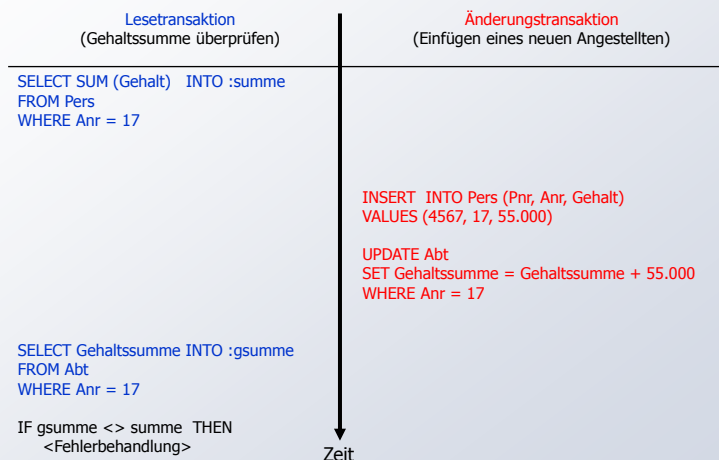
Das wiederholte Lesen einer gegebenen Folge von Daten führt auf verschiedene Ergebnisse:



*Inkonsistenz auch möglich, wenn nicht wiederholt auf das gleiche Datum zugegriffen wird!*

## Phantom-Problem

Einfügungen oder Löschungen können Leser zu falschen Schlussfolgerungen verleiten:



## Integritätsverletzung durch Mehrbenutzer-Anomalie

- Integritätsbedingung:  $A = B$
- $T1 := (A := A + 10; B := B + 10)$   
 $T2 := (A := A * 2; B := B * 2)$
- Probleme bei verschränktem Ablauf

T1	T2	A	B
read (A);		10	10
A := A + 10;		20	
write (A);			
	read (A);		
	A := A * 2;	40	
	write (A);		
	read (B);		20
	B := B * 2;		
	write (B);		
read (B);			30
B := B + 10			
write (B);			

➔ Synchronisation (Sperrungen) einzelner Datensätze reicht nicht aus !

## Cursor-Referenzen

- Zwischen dem Finden eines Objektes mit Eigenschaft P und dem Lesen seiner Daten wird P nach P' verändert

T1	T2
Positioniere Cursor C auf nächstes Objekt (A) mit Eigenschaft P	
	Verändere P → P' bei A
Lies laufendes Objekt	

➔ Cursor-Stabilität sollte gewährleistet werden!

# Synchronisation von Transaktionen

- TRANSAKTION: Ein Programm T mit DML-Anweisungen, das folgende Eigenschaft erfüllt:  
Wenn T **allein** auf einer konsistenten DB ausgeführt wird, dann terminiert T (irgendwann) und hinterlässt die DB in einem konsistenten Zustand. (Während der TA-Verarbeitung gibt es keine Konsistenzgarantien!)

## Ablaufpläne für 3 Transaktionen



- Wenn Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten.
- Ziel der Synchronisation: logischer Einbenutzerbetrieb, d.h. Vermeidung aller Mehrbenutzeranomalien
- Fundamentale Fragestellung: Wann ist die parallele Ausführung von n Transaktionen auf gemeinsamen Daten korrekt?

# Synchronisation von Transaktionen (2)

- Beispiel für einige Ausführungsvarianten (initial:  $A=B=C=10$ )

Ausführung 1		Ausführung 2		Ausführung 3	
T1	T2	T1	T2	T1	T2
read (A)		read (A)		read (A)	
A - 1		A - 1	read (B)	A - 1	
write (A)			B - 2	write (A)	read (B)
read (B)		write (A)			B - 2
B + 1		read (B)	write (B)	read (B)	
write (B)	read (B)	B + 1	read (C)	B + 1	write (B)
	B - 2		C + 2	write (B)	read (C)
	write (B)	write (B)			C + 2
	read (C)		write (C)		write (C)
	C + 2				
	write (C)				

$A=9, B=9, C=12$   
 $A+B+C=30$

dasselbe Ergebnis nach T2; T1

$A=9, B=9, C=12$   
 $A+B+C=30$

$A=9, B=11, C=12$   
 $A+B+C=32$

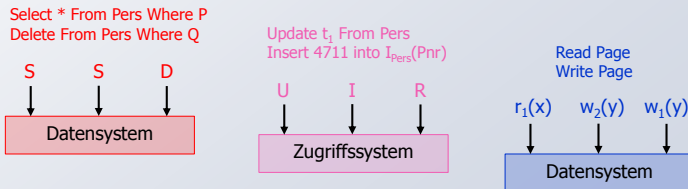
Bei serieller Ausführung bleibt der Wert von  $A + B + C$  unverändert!

Ziel: Äquivalenz der Ergebnisse von verzahnten Ausführungen zu einer der möglichen seriellen Ausführungen

# Modellbildung für die Synchronisation

- Wie kann die Korrektheit der Ausführung im Mehrbenutzerbetrieb überprüft werden?
  - Korrektheitskriterium: Konfliktserialisierbarkeit
  - Geschichtsschreiber zeichnet **Historie H** auf
    - Umformung der aufgezeichneten Operationsfolge H in eine äquivalente serielle Operationsfolge
    - „post mortem“-Analyse
- Tatsächliche Umsetzung
  - Scheduler überprüft jede Operation  $Op_i$  und erzwingt einen serialisierbaren **Ablaufplan S (Schedule)**
    - wenn  $Op_i$  in S konfliktfrei ist, wird sie ausgeführt und an S angehängt
    - sonst wird  $Op_i$  blockiert oder gar die zugehörige Transaktion zurückgesetzt

## Einsatzmöglichkeiten für Geschichtsschreiber oder Scheduler



# Synchronisation - Modellannahmen

- Read/Write-Modell (Page Model)**
  - DB ist Menge von unteilbaren, uninterpretierten Datenobjekten (z. B. Seiten)
 

A

A'
  - DB-Anweisungen lassen sich nachbilden durch atomare Lese- und Schreiboperationen auf Objekten:
    - $r_i[A]$ ,  $w_i[A]$  zum Lesen bzw. Schreiben des Datenobjekts A
    - $c_i$ ,  $a_i$  zur Durchführung eines **commit** bzw. **abort**
  - Transaktion** wird modelliert als eine endliche Folge von Operationen  $p_i$ :
 
$$T = p_1 p_2 p_3 \dots p_n \quad \text{mit} \quad p_i \in \{r[x_i], w[x_i]\}$$
  - Eine vollständige TA hat als letzte Operation entweder einen Abbruch a oder ein Commit c
 
$$T = p_1 \dots p_n a \quad \text{oder} \quad T = p_1 \dots p_n c$$

➔ Für eine TA  $T_i$  werden diese Operationen mit  $r_i$ ,  $w_i$ ,  $c_i$  oder  $a_i$  bezeichnet, um sie zuordnen zu können

## Die Ablauffolge von TA mit ihren Operationen lässt sich wie folgt beschreiben:

$r_1[A] \ r_2[A] \ r_3[B] \ w_1[A] \ w_3[B] \ r_1[B] \ c_1 \ r_3[A] \ w_2[A] \ a_2 \ w_3[C] \ c_3 \dots$

## Korrektheitskriterium der Synchronisation

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

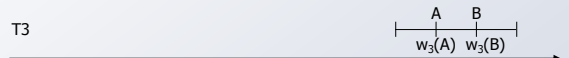
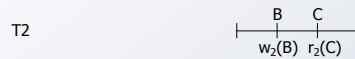
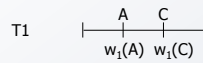
Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Serieller Ablauf von Transaktionen

$TA = \{T1, T2, T3\}$

$DB = \{A, B, C\}$



Ausführungsreihenfolge: T1 | T2 | T3

- n T1 | T2 bedeutet:

**T1 sieht keine Änderungen von T2 und T2 sieht alle Änderungen von T1**

## Korrektheitskriterium der Synchronisation (2)

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Formales Korrektheitskriterium: *Serialisierbarkeit*:

Die parallele Ausführung einer Menge von TA ist **serialisierbar**, wenn es eine serielle Ausführung derselben TA-Menge gibt, die den **gleichen DB-Zustand** und die **gleichen Ausgabewerte** wie die ursprüngliche Ausführung erzielt.

- n Hintergrund:

- Serielle Ablaufpläne sind korrekt!
- Jeder Ablaufplan, der denselben Effekt wie ein serieller erzielt, ist akzeptierbar

## Konsistenzerhaltende Ablaufpläne

- Die TA T1-T3 müssen so synchronisiert werden, dass der resultierende Zustand der DB gleich dem ist, der bei der seriellen Ausführung in einer der folgenden Sequenzen zustande gekommen wäre:

T1, T2, T3      T2, T1, T3      T3, T1, T2  
 T1, T3, T2      T2, T3, T1      T3, T2, T1

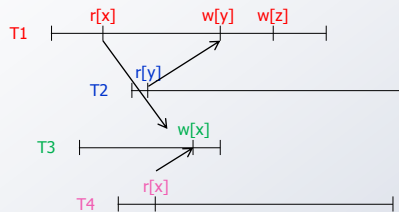
- Bei n TA gibt es n! (hier 3! = 6) mögliche serielle Ablaufpläne
- Serielle Ablaufpläne können verschiedene Ergebnisse haben!

Abbuchung/Einzahlung auf Konto: TA1: - 5000; TA2: + 2000

Konto	Stand = 2000	Limit = 2000
TA1   TA2	4000	(TA1 scheitert am Limit)
TA2   TA1	-1000	

## Konsistenzerhaltende Ablaufpläne (2)

- Nicht alle seriellen Ablaufpläne sind möglich!



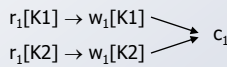
Durch Konfliktoperationen ergeben sich Reihenfolgeabhängigkeiten, die eingehalten werden müssen!

Mögliche Reihenfolgen:

T2 | T1 | T4 | T3  
 T4 | T2 | T1 | T3  
 T2 | T4 | T1 | T3

# Theorie der Serialisierbarkeit

- Ablauf einer Transaktion**
  - Häufigste Annahme: streng sequentielle Reihenfolge der Operationen
  - Serialisierbarkeitstheorie lässt sich auch auf Basis einer partiellen Ordnung ( $<_i$ ) entwickeln
  - TA-Abschluss: **abort** oder **commit** – aber nicht beides!
- Konsistenzanforderungen an eine TA**
  - Falls  $T_i$  ein **abort** durchführt, müssen alle anderen Operationen  $p_i[A]$  vor  $a_i$  ausgeführt werden:  $p_i[A] <_i a_i$
  - Analoges gilt für das **commit**:  $p_i[A] <_i c_i$
  - Wenn  $T_i$  ein Datum A liest und auch schreibt, ist die **Reihenfolge festzulegen**:  
 $r_i[A] <_i w_i[A]$  oder  $w_i[A] <_i r_i[A]$
- Beispiel: Überweisungs-TA T1 (von K1 nach K2)**
  - Totale Ordnung**:  $r_1[K1] \rightarrow w_1[K1] \rightarrow r_1[K2] \rightarrow w_1[K2] \rightarrow c_1$
  - Partielle Ordnung**



# Theorie der Serialisierbarkeit (2)

- Historie<sup>6</sup>**
  - Unter einer Historie versteht man den Ablauf einer (verzahnten) Ausführung mehrerer TA
  - Sie spezifiziert die Reihenfolge, in der die Elementaroperationen verschiedener TA ausgeführt werden
    - Einprozessorsystem: totale Ordnung
    - Mehrprozessorsystem: parallele Ausführung einiger Operationen möglich
- Konfliktoperationen:**

Kritisch sind Operationen verschiedener Transaktionen auf **denselben DB-Daten**, wenn dieser Operationen **nicht reihenfolgeunabhängig** sind!

6. Der Begriff Historie bezeichnet eine retrospektive Sichtweise, also einen abgeschlossenen Vorgang. Ein Scheduling-Algorithmus (Scheduler) produziert Schedules, wodurch noch nicht abgeschlossene Vorgänge bezeichnet werden. Manche Autoren machen jedoch keinen Unterschied zwischen Historie und Schedule.

## Theorie der Serialisierbarkeit (3)

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperreprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Was sind Konfliktoperationen?
  - $r_i[A]$  und  $r_j[A]$ : Reihenfolge ist irrelevant
    - ➔ **kein Konflikt!**
  - $r_i[A]$  und  $w_j[A]$ : Reihenfolge ist relevant und festzulegen.
    - Entweder  $r_i[A] \rightarrow w_j[A]$ 
      - ➔ **R/W-Konflikt!**
    - oder  $w_j[A] \rightarrow r_i[A]$ 
      - ➔ **W/R-Konflikt!**
  - $w_i[A]$  und  $r_j[A]$ : analog
  - $w_i[A]$  und  $w_j[A]$ : Reihenfolge ist relevant und festzulegen
    - ➔ **W/W-Konflikt!**

## Theorie der Serialisierbarkeit (4)

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperreprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Beschränkung auf Konflikt-Serialisierbarkeit<sup>7</sup>
- Historie  $H$  für eine Menge von  $TA \{T_1, \dots, T_n\}$  ist eine Menge von Elementaroperationen mit partieller Ordnung  $<_H$ , so dass gilt:
  1. 
$$H = \bigcup_{i=1}^n T_i \quad (\text{jede } TA \text{ is vollständig in } H \text{ enthalten, ist also abgeschlossen})$$
  2.  $<_H$  ist verträglich mit allen  $<_i$ -Ordnungen, d.h.
 

$$<_H \supseteq \bigcup_{i=1}^n <_i$$

$$\begin{array}{c} r_1[D] \searrow \\ r_1[C] \rightarrow w_1[B] \rightarrow c_1 \\ \downarrow \\ r_2[A] \rightarrow w_2[B] \rightarrow w_2[B] \rightarrow c_2 \end{array}$$

*(lokale Ordnung von ops in  $TA_i$  bleibt erhalten)*
  3. Für zwei Konfliktoperationen  $p, q \in H$  gilt entweder  $p <_H q$  oder  $q <_H p$  *(Reihenfolge von Konfliktop. ist immer definiert)*

Ein Schedule ist ein Präfix einer Historie

7. In der Literatur werden verschiedene Formen der Serialisierbarkeit, also der Äquivalenz zu einer seriellen Historie, definiert. Die **Final-State-Serialisierbarkeit** besitzt die geringsten Einschränkungen. Intuitiv sind zwei Historien (mit der gleichen Menge von Operationen) final-state-äquivalent, wenn sie jeweils denselben Endzustand für einen gegebenen Anfangszustand herstellen. Historien mit dieser Eigenschaft sind in der Klasse FSR zusammengefasst. Die **View-Serialisierbarkeit** (Klasse VSR) schränkt FSR weiter ein. Die hier behandelte **Konflikt-Serialisierbarkeit** (Klasse CSR) ist für praktische Anwendungen die wichtigste. Sie ist effizient überprüfbar und unterscheidet sich bereits dadurch wesentlich von den beiden anderen Serialisierbarkeitsbegriffen. Es gilt:  $CSR \subset VSR \subset FSR$

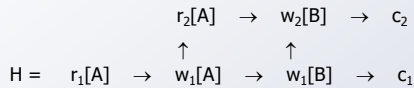


## Theorie der Serialisierbarkeit (5)

### Definition: Äquivalenz zweier Historien

- Zwei Historien  $H$  und  $H'$  sind äquivalent, wenn sie die Konfliktoperationen der nicht abgebrochenen TA in derselben Reihenfolge ausführen:  
 $H \equiv H'$ , wenn  $p_i <_H q_j$ , dann auch  $p_i <_{H'} q_j$
- Anordnung der **konfliktfreien** Operationen ist **irrelevant**
- Reihenfolge der Operationen **innerhalb** einer TA bleibt **invariant**

### Beispiel



- Totale Ordnung

$$H_1 = r_1[A] \rightarrow w_1[A] \rightarrow r_2[A] \rightarrow w_1[B] \rightarrow c_1 \rightarrow w_2[B] \rightarrow c_2$$

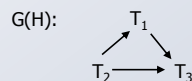
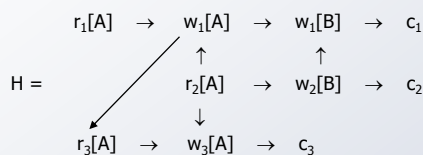
$$H_2 = r_1[A] \rightarrow w_1[A] \rightarrow w_1[B] \rightarrow c_1 \rightarrow r_2[A] \rightarrow w_2[B] \rightarrow c_2$$

$$H_1 \equiv H_2 \text{ (ist seriell)}$$

## Serialisierbare Historie

### Eine Historie $H$ ist serialisierbar, wenn sie äquivalent zu einer seriellen Historie $H_s$ ist

- Einführung eines Konfliktgraph  $G(H)$  (auch Serialisierbarkeitsgraph  $SG(H)$  genannt)
  - Konstruktion des  $G(H)$  über den erfolgreich abgeschlossenen TA
  - Konfliktoperationen  $p_i, q_j$  aus  $H$  mit  $p_i <_H q_j$  fügen eine Kante  $T_i \rightarrow T_j$  in  $G(H)$  ein, falls nicht schon vorhanden
- Beispiel-Historie
- Zugehöriger Konfliktgraph:



### Serialisierbarkeitstheorem

Eine Historie  $H$  ist genau dann serialisierbar, wenn der zugehörige Konfliktgraph  $G(H)$  azyklisch ist

➔ Topologische Sortierung!

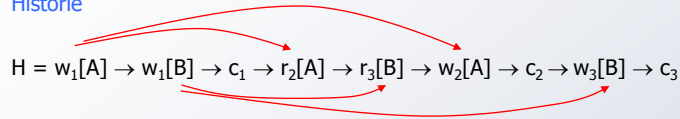
$$T_2 \mid T_1 \mid T_3$$

### CSR

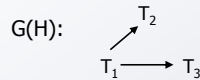
bezeichne die Klasse aller konfliktserialisierbaren Historien. Die Mitgliedschaft in CSR lässt sich in Polynomialzeit in der Menge der teilnehmenden TA testen

## Serialisierbare Historie (2)

Historie



Konfliktgraph



Topologische Ordnungen

$$H_s^1 = w_1[A] \rightarrow w_1[B] \rightarrow c_1 \rightarrow r_2[A] \rightarrow w_2[A] \rightarrow c_2 \rightarrow r_3[B] \rightarrow w_3[B] \rightarrow c_3$$

$$H_s^1 = T_1 \mid T_2 \mid T_3$$

$$H_s^2 = w_1[A] \rightarrow w_1[B] \rightarrow c_1 \rightarrow r_3[B] \rightarrow w_3[B] \rightarrow c_3 \rightarrow r_2[A] \rightarrow w_2[A] \rightarrow c_2$$

$$H_s^2 = T_1 \mid T_3 \mid T_2$$

$$H \equiv H_s^1 \equiv H_s^2$$

## Serialisierbare Historie (3)

Anforderungen an im DBMS zugelassene Historien

- Serialisierbarkeit ist eine Minimalanforderung
- TA  $T_j$  sollte zu jedem Zeitpunkt vor Commit **lokal rücksetzbar** sein
  - andere mit Commit abgeschlossene  $T_i$  dürfen nicht betroffen sein
  - kritisch sind Schreib-/Leseabhängigkeiten  $w_i[A] \rightarrow \dots \rightarrow r_i[A]$
- Wie kritisch für das lokale Rücksetzen von  $T_j$  sind
  - $r_i[A] \rightarrow \dots \rightarrow w_j[A]$
  - oder
  - $w_j[A] \rightarrow \dots \rightarrow w_i[A]$
  - oder
  - $w_i[A] \rightarrow \dots \rightarrow w_j[A]$

Serialisierbarkeitstheorie: Gebräuchliche Klassenbeziehungen<sup>9</sup>

- SR: serialisierbare Historien
- RC: rücksetzbare Historien
- ACA: Historien ohne kaskadierendes Rücksetzen
- ST: strikte Historien

9. Weikum, G., Vossen, G.: Transactional Information Systems, Morgan Kaufmann, 2001, unterscheidet unter Berücksichtigung von VSR und FSR 10 Klassen von serialisierbaren Historien.

## Rücksetzbare Historie

Definition:  $T_i$  liest von  $T_j$  in  $H$ , wenn gilt

- $T_j$  schreibt mindestens ein Datum  $A$ , das  $T_i$  nachfolgend liest:  
 $w_j[A] <_H r_i[A]$
- $T_j$  wird (zumindest) nicht vor dem Lesevorgang von  $T_i$  zurückgesetzt:  
 $a_j <_H r_i[A]$
- Alle anderen zwischenzeitlichen Schreibvorgänge auf  $A$  durch andere TA  $T_k$  werden vor dem Lesen durch  $T_i$  zurückgesetzt.

Falls  
 $w_j[A] <_H w_k[A] <_H r_i[A]$ ,

muss auch  
 $a_k <_H r_i[A]$  gelten.

$H = \dots w_j[A] \rightarrow \dots \rightarrow w_k[A] \rightarrow \dots a_k \rightarrow \dots \rightarrow r_i[A]$

Definition: Eine **Historie  $H$  heißt rücksetzbar**, falls immer die schreibende TA ( $T_j$ ) vor der lesenden TA ( $T_i$ ) ihr Commit ausführt:

$c_j <_H c_i$

$H = \dots w_j[A] \rightarrow r_i[A] \rightarrow w_i[B] \rightarrow c_j \rightarrow \dots \rightarrow a_i [c_i]$

## Historie ohne kaskadierendes Rücksetzen

Kaskadierendes Rücksetzen

Schritt	T1	T2	T3	T4	T5
0.	...				
1.	$w_1[A]$				
2.		$r_2[A]$			
3.		$w_2[B]$			
4.			$r_3[B]$		
5.			$w_3[C]$		
6.				$r_4[C]$	
7.				$w_4[D]$	
8.					$r_5[D]$
9.	$a_1$ (abort)				

➔ In der Theorie lässt sich ACID garantieren! Aber ...

Definition: Eine **Historie vermeidet kaskadierendes Rücksetzen**, wenn

$c_j <_H r_i[A]$

gilt, wann immer  $T_i$  ein von  $T_j$  geändertes Datum liest

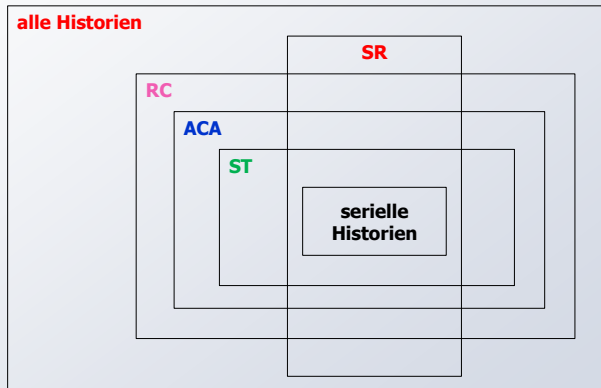
➔ **Änderungen dürfen erst nach Commit freigegeben werden!**

## Klassen von Historien

- Definition: Eine **Historie H ist strikt**, wenn für je zwei TA  $T_i$  und  $T_j$  gilt:

Wenn  $w_j[A] <_H o_i[A]$  (mit  $o_i = r_i$  oder  $o_i = w_i$ ),  
dann muss gelten:  
 $c_j <_H o_i[A]$  oder  $a_j <_H o_i[A]$

- Beziehungen zwischen den Klassen



➔ Schlussfolgerungen ?

## Klassen von Historien (2)

- Beispiele

$$r_i[C] \rightarrow w_j[B] \rightarrow r_i[A] \rightarrow c_i$$

H:  $r_j[B] \rightarrow w_j[B] \rightarrow w_j[A] \rightarrow c_j$

$H_{SR}$ :  $r_i[C] \rightarrow r_j[B] \rightarrow w_j[B] \rightarrow w_i[B] \rightarrow w_j[A] \rightarrow r_i[A] \rightarrow c_i \rightarrow c_j$

$H_{RC}$ :  $r_i[C] \rightarrow r_j[B] \rightarrow w_j[B] \rightarrow w_i[B] \rightarrow w_j[A] \rightarrow r_i[A] \rightarrow c_j \rightarrow c_i$

$H_{ACA}$ :  $r_i[C] \rightarrow r_j[B] \rightarrow w_j[B] \rightarrow w_i[B] \rightarrow w_j[A] \rightarrow c_j \rightarrow r_i[A] \rightarrow c_i$

$H_{ST}$ :  $r_i[C] \rightarrow r_j[B] \rightarrow w_j[B] \rightarrow w_j[A] \rightarrow c_j \rightarrow w_i[B] \rightarrow r_i[A] \rightarrow c_i$

$H_S$ :  $r_j[B] \rightarrow w_j[B] \rightarrow w_j[A] \rightarrow c_j \rightarrow r_i[C] \rightarrow w_i[B] \rightarrow r_i[A] \rightarrow c_i$

- Scheduler gewährleistet die Einhaltung der Konfliktserialisierbarkeit der gewählten Klasse

- hier nur Diskussion einfacher Sperrverfahren
- Scheduler heißt Sperrverwalter oder Lock Manager

## RX-Sperrverfahren

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Sperrmodi
  - Sperrmodus des Objektes: NL (no lock), R (read), X (exclusive)
  - Sperranforderung einer Transaktion: R, X
- Kompatibilitätsmatrix:

		aktueller Modus des Objekts		
		NL	R	X
angeforderter Modus der TA	R	+	+	-
	X	+	-	-

- Falls Sperre nicht gewährt werden kann, muss die **anfordernde TA warten, bis das Objekt freigegeben wird** (Commit/Abort der die Sperre besitzenden TA)
- Wartebeziehungen werden in einem **Wait-for-Graph (WfG)** verwaltet

## RX-Sperrverfahren (2)

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Ablauf von Transaktionen

T1	T2	a	b	Bem.
		NL	NL	
lock(a, X)		X <sub>1</sub>		
...				
	lock(b, R)		R <sub>2</sub>	
	...			
lock(b, R)			R <sub>2</sub> , R <sub>1</sub>	
	lock(a, R)	X <sub>1</sub>		T2 wartet, WfG: T <sub>2</sub> $\xrightarrow{a}$ T <sub>1</sub>
...				
unlock(a)		NL $\rightarrow$ R <sub>2</sub>		T2 wecken, WfG: -
...	...			
unlock(b)			R <sub>2</sub>	

# Zweiphasen-Sperrprotokolle<sup>10</sup>

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

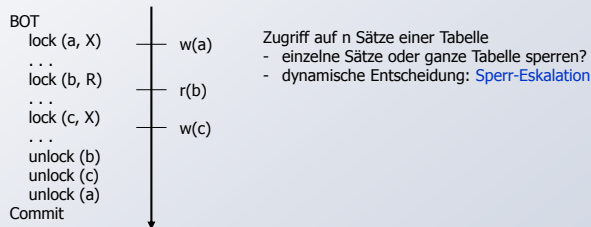
Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Einhaltung folgender Regeln gewährleistet Serialisierbarkeit:
  - Vor jedem Objektzugriff muss Sperre mit ausreichendem Modus angefordert werden
  - Gesetzte Sperren anderer TA sind zu beachten
  - Eine TA darf nicht mehrere Sperren für ein Objekt anfordern
  - Zweiphasigkeit:**
    - Anfordern von Sperren erfolgt in einer *Wachstumsphase*
    - Freigabe der Sperren in *Schrumpfungsphase*
    - Sperrfreigabe kann erst beginnen, wenn alle benötigten Sperren gehalten werden
  - Spätestens bei Commit sind alle Sperren freizugeben
- Beispiel für ein 2PL-Protokoll (2PL: two-phase locking)



An der SQL-Schnittstelle ist die Sperranforderung und -freigabe nicht sichtbar!

10. Eswaran, K.P. et al.: The notions of consistency and predicate locks in a data base system, in: Comm. ACM 19:11, 1976, 624-633

# Zweiphasen-Sperrprotokolle (2)

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Anwendung des 2PL-Protokolls

T1	T2	Bem.
BOT		
lock (a,X)		
read (a)		
write(a)		
	BOT	
	lock (a, X)	T2 wartet, WfG: $T_2 \xrightarrow{a} T_1$
lock (b, X)		
read (b)		
unlock (a)		T2 wecken, WfG: -
	read (a)	
	write (a)	
	unlock (a)	
	commit	
unlock (b)		
<b>abort!</b>		<b>dirty read!</b>

➔ Zweiphasiges Protokoll reicht für den praktischen Einsatz nicht aus !

## Zweiphasen-Sperrprotokolle (3)

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

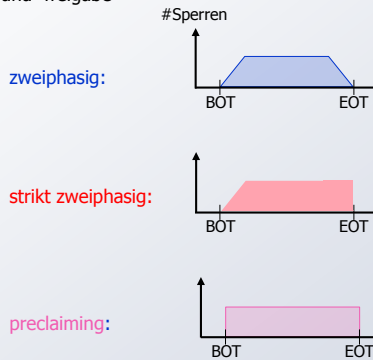
Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Formen der Zweiphasigkeit
- Sperranforderung und -freigabe



- Strikte 2PL-Protokolle
  - SS2PL** (strong 2PL) gibt alle Sperren (X und R) erst bei Commit frei
  - S2PL** (strict 2PL) gibt alle X-Sperren erst bei Commit frei
  - Sie **verhindern dadurch kaskadierendes Rücksetzen**

## Verklemmungen (Deadlocks)

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

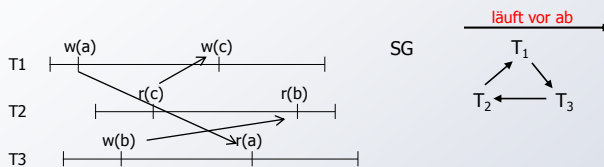
Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

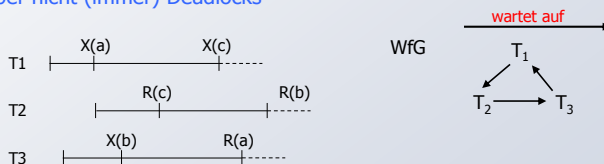
Zwei-Phasen-Commit-Protokoll (2PC)

- Auftreten von Verklemmungen ist **inhärent** und kann bei pessimistischen Methoden (blockierende Verfahren) nicht vermieden werden.

- Nicht-serialisierbare Historie



- RX-Verfahren verhindert das Auftreten einer nicht-serialisierbaren Historie, aber nicht (immer) Deadlocks



# Logging und Recovery<sup>11</sup>

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

- Aufgabe des DBMS:
  - Automatische Behandlung aller erwarteten Fehler
- Was sind erwartete Fehler?<sup>12</sup>
  - DB-Operation wird zurückgewiesen, Commit wird nicht akzeptiert, . . .
  - Stromausfall, DBMS-Probleme, . . .
  - Geräte funktionieren nicht (Spur, Zylinder, Platte defekt)
  - auch beliebiges Fehlverhalten der Gerätesteuerung?
  - falsche Korrektur von Lesefehlern? . . .
- Fehlermodell von zentralisierten DBMS
  - Transaktionsfehler (z. B. Deadlock) → Transaktions-Recovery
  - Systemfehler (Verlust aller HSP-Inhalte) → Crash-Recovery
  - Gerätefehler → Medien-Recovery
  - Katastrophen → Katastrophen-Recovery
- Erhaltung der physischen Datenintegrität
  - Periodisches Erstellen von Datenkopien
  - Führen von Änderungsprotokollen für den Fehlerfall (Logging)<sup>13</sup>
  - Bereitstellen von Wiederherstellungsalgorithmen im Fehlerfall (Recovery)

11. Härder, T., Reuter, A.: Principles of Transaction Oriented Database Recovery, in: ACM Computing Surveys 15:4, Dec. 1983, 287-317.  
 12. Kommerzielle Anwendungen auf Großrechnern sind durch ihre Zuverlässigkeit gekennzeichnet. Nicht selten besteht der Code bis zu 90% aus (erprobten) Recovery-Routinen (W. G. Spruth).  
 13. Commercial DB systems spend almost 2/3 of their total elapsed time inside the logging system, and locking is also another non-trivial bottleneck (Stonebraker et al., 2007)

# Was passiert bei einem Crash?

Atomarität von DB-Operationen und Trans-Aktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

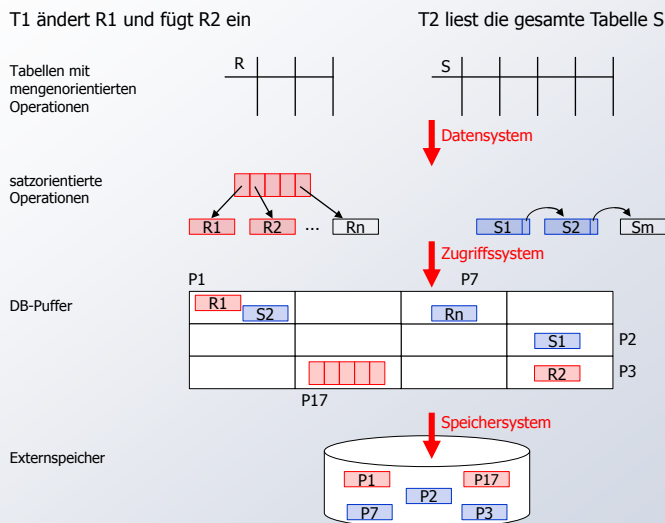
Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperprotokolle (2PL)

Logging und Recovery

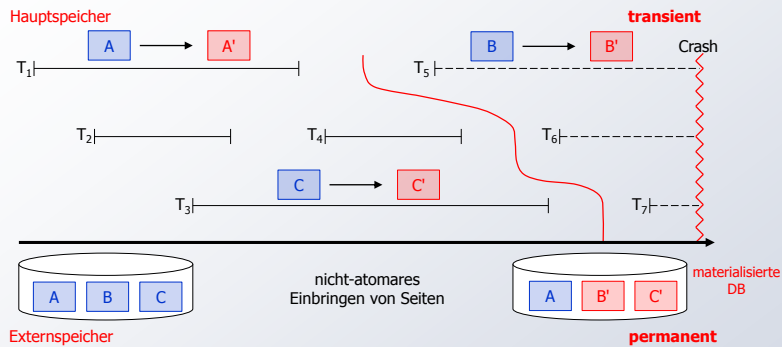
Zwei-Phasen-Commit-Protokoll (2PC)



- Logging
  - Sammeln von Redundanz im Normalbetrieb zur Fehlerbehebung
  - Verschiedene Verfahren für Logging
  - Logisch, physisch, physiologisch

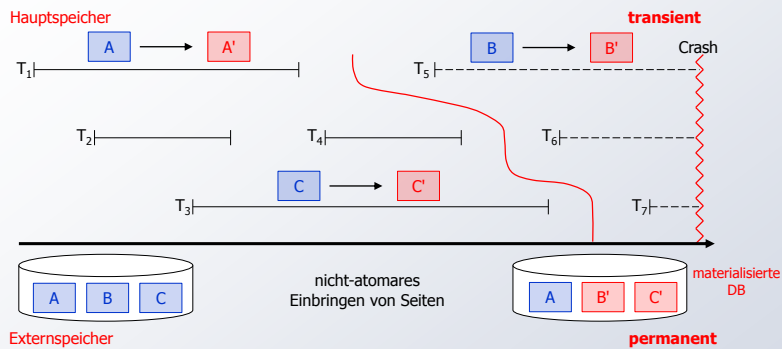


## Logging und Recovery (2)



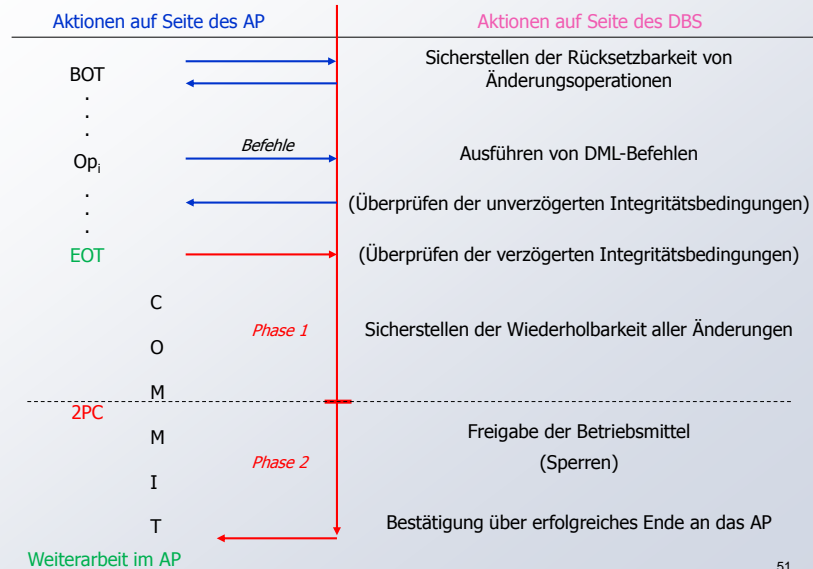
- DBMS garantiert physische Datenintegrität
  - Bei jedem Fehler (z. B. Ausfall des Rechners, Crash des Betriebssystems oder des DBMS, Fehlerhaftigkeit einzelner Transaktionsprogramme) wird eine „korrekte“ Datenbank rekonstruiert
  - Nach einem (Teil-)Crash ist immer der jüngste transaktionskonsistente Zustand der DB zu rekonstruieren, in dem alle Änderungen von Transaktionen enthalten sind, die vor dem Zeitpunkt des Fehlers erfolgreich beendet waren ( $T_1$  bis  $T_4$ ) und sonst keine
  - automatische Wiederherstellung bei Restart (Wiederanlauf) des Systems

## Logging und Recovery (3)



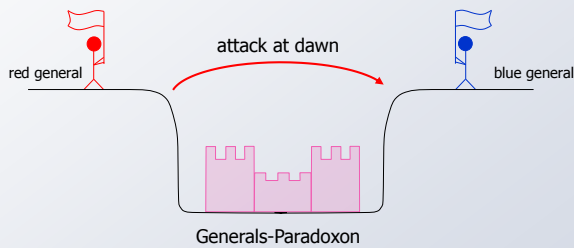
- Maßnahmen beim Wiederanlauf (siehe auch Beispiel)
  - Ermittlung der beim Crash aktiven Transaktionen ( $T_5, T_6, T_7$ )
  - Wiederholen (REDO) der Änderungen von abgeschlossenen Transaktionen, die vor dem Crash nicht in die Datenbank zurückgeschrieben waren ( $A \rightarrow A'$ )
  - Rücksetzen (UNDO) der Änderungen der aktiven Transaktionen in der Datenbank ( $B' \rightarrow B$ )

## Schnittstelle zwischen AP und DBS – transaktionsbezogene Aspekte



## Verarbeitung in Verteilten Systemen

- Ein *verteiltes System* besteht aus autonomen Subsystemen, die koordiniert zusammenarbeiten, um eine gemeinsame Aufgabe zu erfüllen
  - Client/Server-Systeme
  - Mehrrechner-DBS, . . .
- Beispiel: The „Coordinated Attack“ Problem



### Grundproblem verteilter Systeme

Das für verteilte Systeme charakteristische Kernproblem ist der Mangel an globalem (zentralisiertem) Wissen

- symmetrische Kontrollalgorithmen sind oft zu teuer oder zu ineffektiv
- fallweise Zuordnung der Kontrolle

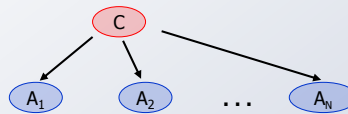
## Verarbeitung in Verteilten Systemen (2)

### Erweitertes Transaktionsmodell

**verteilte** Transaktionsbearbeitung (Primär-, Teiltransaktionen) – **zentralisierte Steuerung** des Commit-Protokolls

1 Koordinator

N Teiltransaktionen (Agenten)



➔ *rechnerübergreifendes Mehrphasen-Commit-Protokoll notwendig, um Atomarität einer globalen Transaktion sicherzustellen*

### Anforderungen an geeignetes Commit-Protokoll:

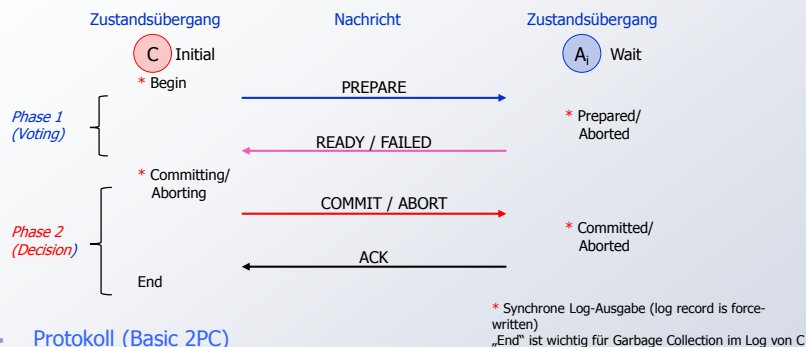
- Geringer Aufwand (#Nachrichten, #Log-Ausgaben)
  - Minimale Antwortzeitverlängerung (Nutzung von Parallelität)
  - Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern
- ➔ **Zentralisiertes Zweiphasen-Commit-Protokoll stellt geeignete Lösung dar**

### Erwartete Fehlersituationen

- Transaktionsfehler
- Systemfehler (Crash)
- ➔ **i. allg. partielle Fehler (Rechner, Verbindungen, ...)**
- Gerätefehler

➔ Fehlererkennung z. B. über Timeout

## Zentralisiertes Zweiphasen-Commit



### Protokoll (Basic 2PC)

- Folge von Zustandsänderungen für Koordinator und für Agenten
  - Protokollzustand auch nach Crash eindeutig: synchrones Logging
  - Sobald C ein NO VOTE (FAILED) erhält, entscheidet er ABORT
  - Sobald C die ACK-Nachricht von allen Agenten bekommen hat, weiß er, dass alle Agenten den Ausgang der TA kennen

➔ **C kann diese TA vergessen, d. h. ihre Log-Einträge im Log löschen!**

- Warum ist das 2PC-Protokoll blockierend?

### Aufwand im Erfolgsfall:

- Nachrichten: 4N
- Log-Ausgaben (forced log writes): 2 + 2N

## Commit: Kostenbetrachtungen

- vollständiges 2PC-Protokoll  
( $N = \# \text{Teil-TA}$ , davon  $M = \# \text{Leser}$ )
  - Nachrichten:  $4N$
  - Log-Ausgaben:  $2 + 2N$
  - Antwortzeit:  
längste Runde in Phase 1 (kritisch, weil Betriebsmittel blockiert)  
+ längste Runde in Phase 2
- Aufwand bei spezieller Optimierung für Leser:  
Lesende Teil-TA nehmen nur an Phase 1 teil, dann Freigabe der Sperren
  - Nachrichten:  
 $4N - 2M$
  - Log-Ausgaben:  $2 + 2N - 2M$   
für  $N > M$
- Lässt sich das zentralisierte 2PC-Protokoll weiter optimieren?

## Zusammenfassung

- Transaktionsparadigma
  - Verarbeitungsklammer für die Einhaltung der Constraints des DB-Schemas
  - Verdeckung der Nebenläufigkeit (*concurrency isolation*)
    - ➔ Synchronisation
  - Verdeckung von (erwarteten) Fehlerfällen (*failure isolation*)
    - ➔ Logging und Recovery
- Erhaltung der semantischen Datenintegrität
  - Beschreibung der „Richtigkeit“ von Daten durch Prädikate und Regeln
  - Merke: „Ohne Ziel ist jeder Schuss ein Treffer“ oder „ohne Spezifikation von Constraints ist jeder Zustand richtig“
  - Unterscheide: Transaktionskonsistenz, Operationskonsistenz, Aktionskonsistenz, Gerätekonsistenz
- Beim ungeschützten und konkurrierenden Zugriff von Lesern und Schreibern auf gemeinsame Daten können Anomalien auftreten
- Theorie der Serialisierbarkeit
  - **Konfliktoperationen:**  
Kritisch sind Operationen verschiedener Transaktionen auf **denselben DB-Daten**, wenn diese Operationen **nicht reihenfolgeunabhängig** sind!
  - **Serialisierbarkeitstheorem:**  
Eine Historie  $H$  ist genau dann serialisierbar, wenn der zugehörige Konfliktgraph  $G(H)$  azyklisch ist

Atomarität von DB-Operationen und Transaktionen?

Erhaltung der DB-Konsistenz

Anomalien im Mehrbenutzerbetrieb

Synchronisation von Transaktionen

Theorie der Serialisierbarkeit

Zwei-Phasen-Sperrprotokolle (2PL)

Logging und Recovery

Zwei-Phasen-Commit-Protokoll (2PC)

© 2014 LG IS

- **Serialisierbare Abläufe**
  - gewährleisten „automatisch“ Korrektheit des Mehrbenutzerbetriebs
  - erzwingen u. U. lange Blockierungszeiten paralleler Transaktionen
- **Realisierung der Synchronisation durch Sperrverfahren**
  - Sperren stellen während des laufenden Betriebs sicher, dass die resultierende Historie serialisierbar bleibt
  - Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt (Deadlock-Problem ist inhärent)
  - Basis-Protokoll: **2PL**
    - ➔ Sperrverfahren sind pessimistisch und universell einsetzbar
- **Logging- und Recovery-Verfahren**
  - automatische Behandlung für erwartete Fehler
  - Fehlerarten: Transaktions-, System-, Gerätefehler und Katastrophen
- **Zweiphasen-Commit-Protokolle (2PC)**
  - Hoher Aufwand an Kommunikation und E/A
  - Optimierungsmöglichkeiten sind zu nutzen
  - Maßnahmen erforderlich, um Blockierungen zu vermeiden!
    - ➔ Kritische Stelle: Ausfall von C