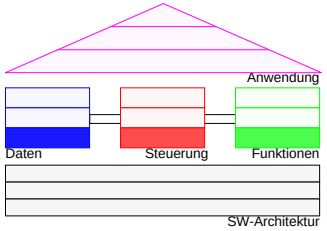
6. Grundlagen des Transaktionskonzepts

- **GBIS-Rahmen: Einordnung**



- **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**
  - Logging kann schichtenspezifisch gewählt werden
  - Einbringverfahren garantiert bei Crash entsprechende DB-Konsistenz
- **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 Operatonen im Mehrbenutzerbetrieb
  - vielfältige Fehler im DBS und seiner Umgebung

  ➥ Logging und Recovery bietet Schutz vor **erwarteten Fehlern**!
- **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):[1]
    - für Web-Sites: 99%
    - für gut administrierte Systeme: 99,99%
    - höchstens: 99,999%
  - Künftige Verfügbarkeit
    - da fehlen noch 5 9’
    - bis 2010 nicht zu erreichen???
    - . . .

1. 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, TAS)

Bausteine für Transaktionen – Atomare Aktionen

Schichtenspezifische Operationen

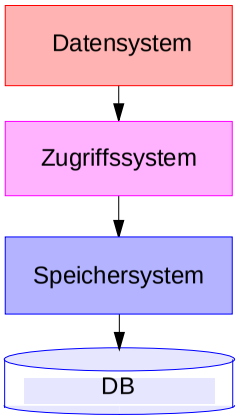
Select ... From ... Where
Insert ... Into

Füge Satz ein
Modifiziere Zugriffspfad

Stelle Seite bereit
Gib Seite frei

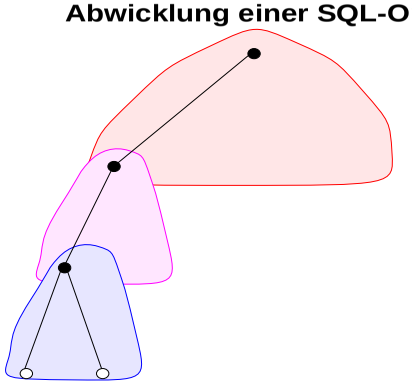
Lies / Schreibe Seite

Atomare Aktionen und Benutzthierarchie



Gedankenversuch

Abwicklung einer SQL-Op



Auch Atomare Aktionen sind Abstraktionen!

Bausteine für Transaktionen – Atomare Aktionen (2)

Schichtenspezifische Operationen

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

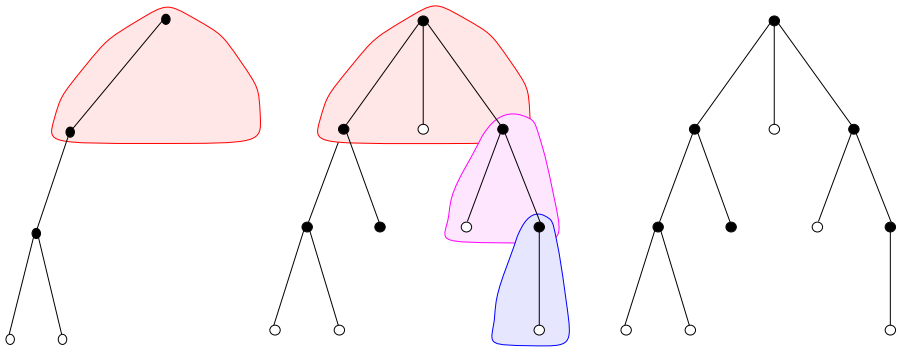
Füge Satz ein
Modifiziere Zugriffspfad

Stelle Seite bereit
Gib Seite frei

Lies / Schreibe Seite

Gedankenversuch

Abwicklung einer SQL-Op



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

Schutzbedürfnis einer flachen Transaktion und Zusicherungen an den Programmierer

Schutzmaßnahmen im Ausführungspfad des DBS funktionieren nicht!

Schichtenspezifische Operationen

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 transarency)

Erhaltung der DB-Konsistenz

- **Abbildung der Miniwelt**

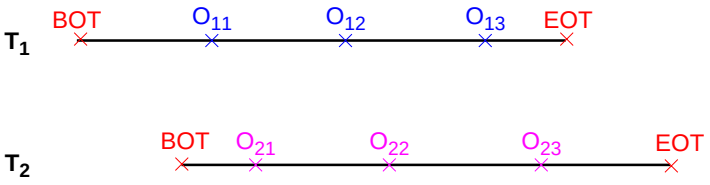
- **Erhaltung der semantischen Datenintegrität**
  - Beschreibung der „Richtigkeit“ von Daten durch Prädikate und Regeln, bereits bekannt:
    - modellinhärente Bedingungen (relationale Invarianten)
    - anwendungsspezifische Bedingungen (Check, Unique, Not Null, ...)
  - aktive Maßnahmen des DBS erwünscht (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.*

Erhaltung der DB-Konsistenz (2)

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



BOT: Begin of Transaction EOT (Commit): End of Transaction
O_{ij} : DB-Operation; Lese- und Schreiboperationen auf DB-Daten

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

- **Verfeinerung des Konsistenzbegriffes**
  - Transaktionsatomarität impliziert Transaktionskonsistenz: nur Änderungen erfolgreicher Transaktionen sind in der DB enthalten
  - Anweisungsatomarität impliziert DML-Operationskonsistenz[2]: DML-Operation hält schichtenspezifische Konsistenz des Datensystems ein
  - Wegen des hierarchischen Aufbaus von DML-Operationen aus atomaren Aktionen setzt DML-Operationskonsistenz Aktionskonsistenz voraus
  - Geräte-/Dateikonsistenz ist wiederum Voraussetzung, dass Aktionen überhaupt auf den Daten abgewickelt werden können

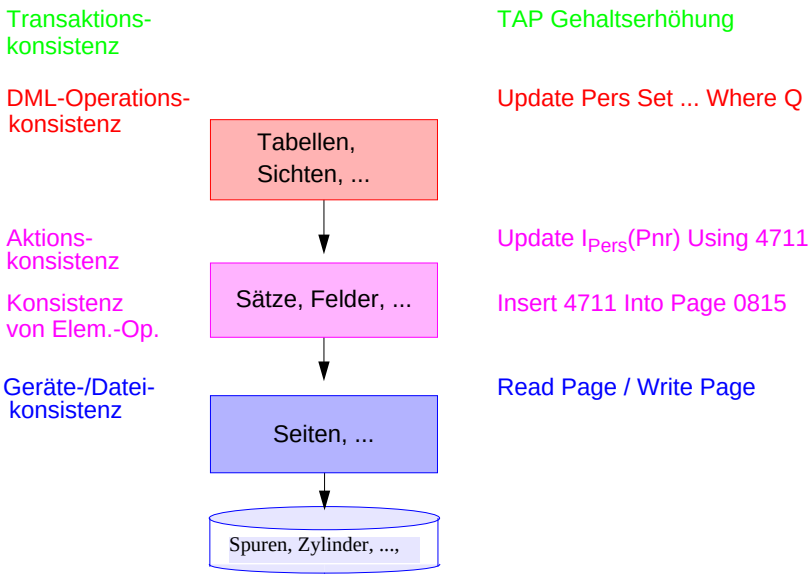
➥ Konsistenz einer Schicht setzt schichtenspezifische Konsistenz aller darunter liegenden Schichten voraus!

2. **„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 (3)

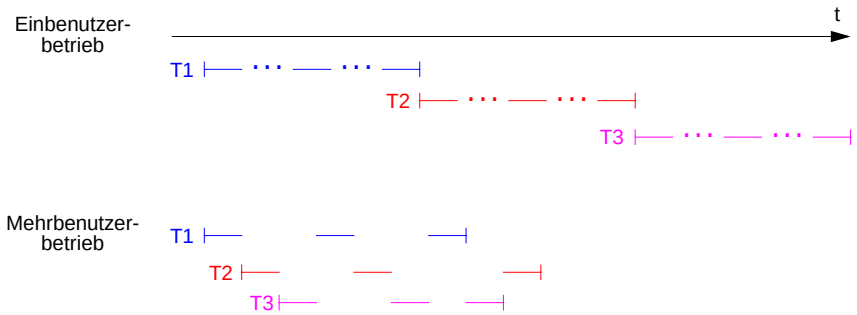
- **Welche Konsistenzart garantiert jede Schicht nach erfolgreichem Abschluss einer schichtenspezifischen Operation?**
  - Speichersystem → Geräte-/Dateikonsistenz (einzelne Seite)
    Jede Seite muss physisch unversehrt, d. h. lesbar oder schreibbar sein
  - Zugriffssystem → Aktionskonsistenz (mehrere Seiten)
    Sätze und Zugriffspfade müssen für Aktionen „in sich konsistent“ sein, d. h. beispielsweise: „Alle Zeiger müssen stimmen!“
    Sonderfall: Konsistenz von Elementaroperationen (einzelne Seite)
  - Datensystem → DML-Operationskonsistenz (oft viele Seiten)
  - Datenbank → Transaktionskonsistenz
    Alle Constraints des DB-Schemas müssen erfüllt sein!

Systemhierarchie + DB-Konsistenz



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**

Unkontrollierter Mehrbenutzerbetrieb

- **Abhängigkeit von nicht freigegebenen Änderungen**

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

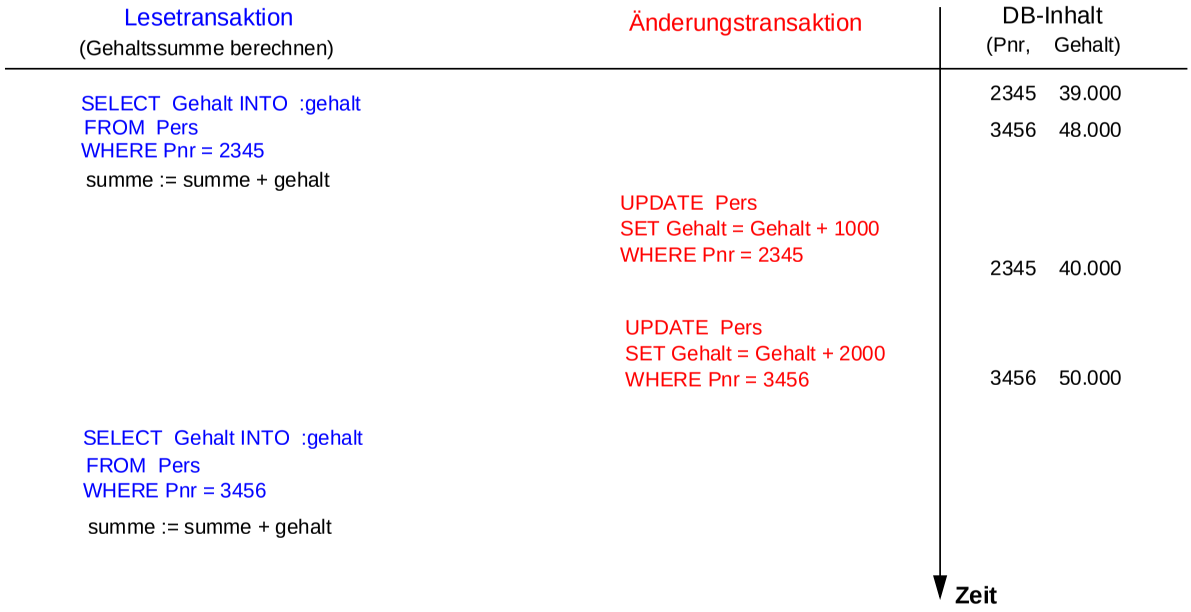
- **Verlorengegangene Änderung (Lost Update)**

T1	T2	A in DB
read (A);		
	read (A);	
A := A - 1;		
write (A);		
	A := A - 1;	
	write (A);	

➥ **Verlorengegangene Änderungen sind auszuschließen!**

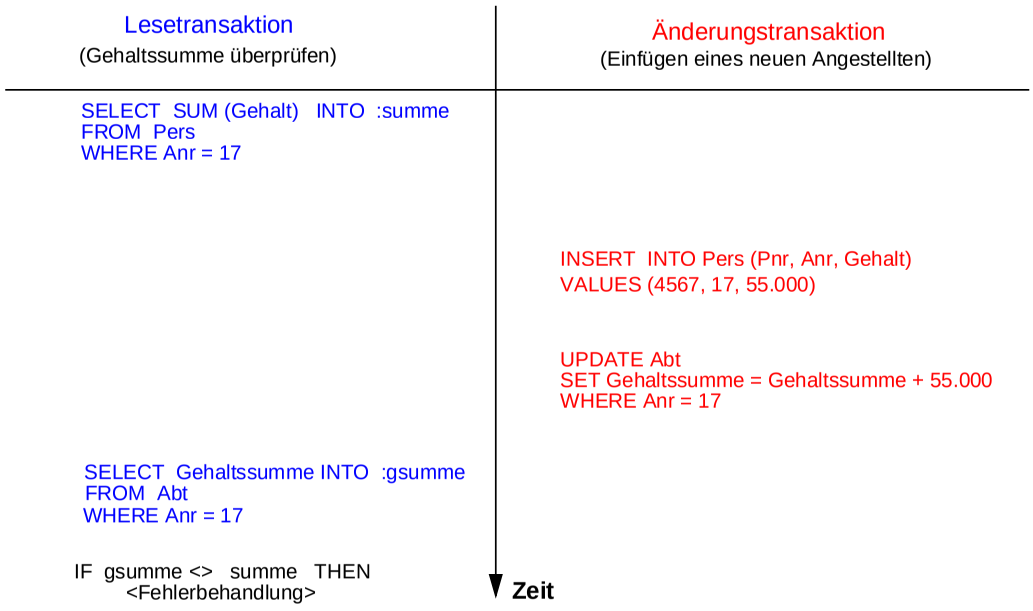
Inkonsistente Analyse (Non-repeatable Read)

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



Phantom-Problem

Einfügungen oder Löschungen können Leser zu falschen Schlussfolgerungen verleiten:



Unkontrollierter Mehrbenutzerbetrieb (2)

- **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); A := A + 10; write (A);			
	read (A); A := A ∗ 2; write (A); read (B) B := B ∗ 2; write (B);		
read (B); B := B + 10; write (B);			

➥ **Synchronisation (Sperren) 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	

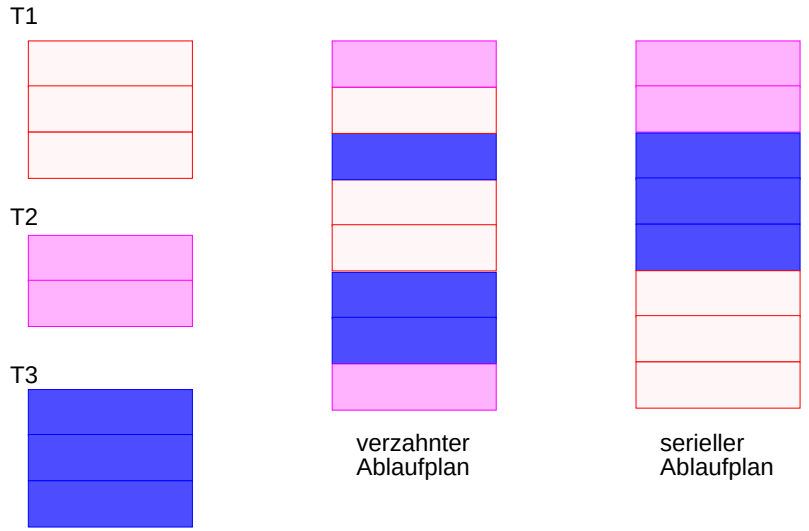
➥ **Cursor-Stabilität sollte gewährleistet werden!**

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**

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

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

- **Was ist das Ergebnis der verschiedenen Ausführungsvarianten?**

	A	B	C	A + B + C
initialer Wert				
nach T1; T2				
nach Ausf. 2				
nach Ausf. 3				
nach T2; T1				

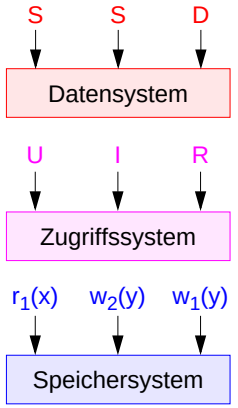
- **Ziel:** Äquivalenz der Ergebnisse von verzahnten Ausführungen zu einer der möglichen seriellen Ausführungen

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**



Select * From Pers Where P
Delete From Pers Where Q

Update t_1 From Pers
Insert 4711 into I_{Pers}(Pnr)

Read Page
Write Page

Synchronisation – Modellannahmen

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

    $T = p_1\ p_2\ p_3\ ...\ p_n$ mit $p_i \in \{r[x_i], w[x_i]\}$

  - Eine vollständige TA hat als letzte Operation entweder einen Abbruch a oder ein Commit c

    $T = p_1\ ...\ p_n\ a$ oder $T = p_1\ ...\ p_n\ c$

➥ **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:**

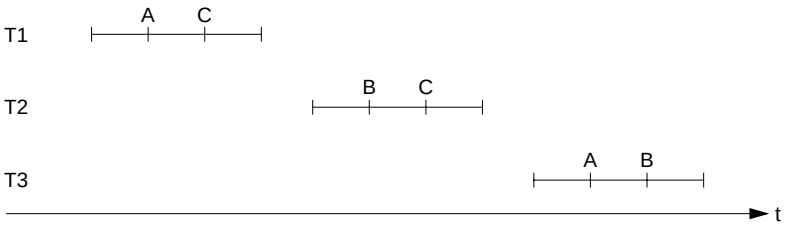
$r_1[A]\ r_2[A]\ r_3[B]\ w_1[A]\ w_3[B]\ r_1[B]\ c_1\ r_3[A]\ w_2[A]\ a_2\ w_3[C]\ c_3\ ...$

Korrektheitskriterium der Synchronisation

- **Serieller Ablauf von Transaktionen**

TA = {T1, T2, T3}

DB = {A, B, C}



Ausführungsreihenfolge:

- **T1 | T2 bedeutet:**

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

- **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.

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

Konsistenzerhaltende Ablaufpläne

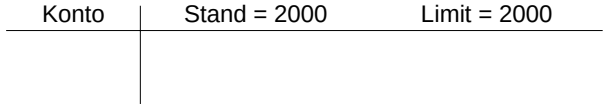
- Die TA T1-T3 müssen so synchronisiert werden, dass der resultierende Zustand der DB gleich dem ist, der bei der seriellen Ausführung in einer der folgenden Sequenzen zustande gekommen wäre:

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

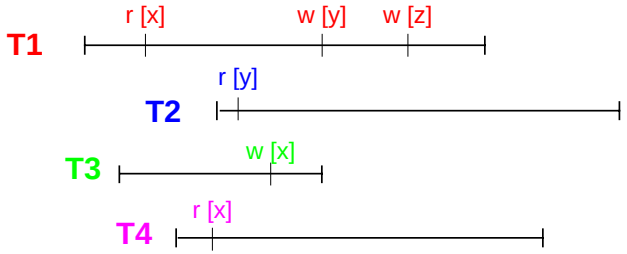
- **Bei n TA gibt es n! (hier 3! = 6) mögliche serielle Ablaufpläne**

- **Serielle Ablaufpläne können verschiedene Ergebnisse haben!**

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



- **Nicht alle seriellen Ablaufpläne sind möglich!**



- **Sinnvolle Einschränkungen**

1. **Reihenfolgeerhaltende Serialisierbarkeit:**

   Jede TA sollte wenigstens alle Änderungen sehen, die bei ihrem Start (BOT) bereits beendet waren

2. **Chronologieerhaltende Serialisierbarkeit:**

   Jede TA sollte stets die aktuellste Objektversion sehen

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!

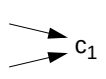
- **Konsistenzanforderungen an eine TA**
  - Falls T_i ein **abort** durchführt, müssen alle anderen Operationen $p_i[A]$ vor a_i ausgeführt werden: $p_i[A] <_i a_i$
  - Analoges gilt für das **commit:** $p_i[A] <_i c_i$
  - Wenn T_i ein Datum A liest und auch schreibt, ist die **Reihenfolge festzulegen**:

    $r_i[A] <_i w_i[A]$ oder $w_i[A] <_i r_i[A]$

- **Beispiel: Überweisungs-TA T1** (von K1 nach K2)

$r_1[K1]$	oder	$r_1[K1]$	$r_1[K2]$
$w_1[K1]$		$w_1[K1]$	$w_1[K2]$
$r_1[K2]$			c_1
$w_1[K2]$			
c_1			

  - 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**[3]
  - 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 ➥ partielle Ordnung

- **Konfliktoperationen:**

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

- **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!**

3. 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)

- **Beschränkung auf Konflikt-Serialisierbarkeit**[4]

- **Historie H für eine Menge von TA {T1, ..., Tn}**
  ist eine Menge von Elementaroperationen mit partieller Ordnung $<_H$,
  so dass gilt:

  1. $H = \bigcup_{i=1}^{n} T_i$

  2. $<_H$ ist verträglich mit allen $<_i$-Ordnungen, d.h.

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

  3. Für zwei Konfliktoperationen $p, q \in H$ gilt entweder

     $p <_H q$

     oder

     $q <_H p$

- **Ein Schedule ist ein Präfix einer Historie**

4. 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 (4)

- **Beispiel-Historie für 3 TA**

$$
\begin{array}{lccccccccc}
 & & & & r_2[A] & \rightarrow & w_2[B] & \rightarrow & w_2[C] & \rightarrow & c_2 \\
 & & & & \uparrow & & \uparrow & & \uparrow & & \\
H = & r_3[B] & \rightarrow & w_3[A] & \rightarrow & w_3[B] & \rightarrow & w_3[C] & \rightarrow & c_3 & \\
 & & & \uparrow & & & & & & & \\
 & r_1[A] & \rightarrow & w_1[A] & \rightarrow & c_1 & & & & &
\end{array}
$$

- Reihenfolge konfliktfreier Operationen (zwischen TA) wird nicht spezifiziert

- Mögliche totale Ordnung[5]

$$H_1 = r_1[A] \rightarrow r_3[B] \rightarrow w_1[A] \rightarrow w_3[A] \rightarrow c_1 \rightarrow r_2[A] \rightarrow w_3[B] \rightarrow$$

$$w_3[C] \rightarrow c_3 \rightarrow w_2[B] \rightarrow w_2[C] \rightarrow c_2$$

5. Alternative Schreibweise bei einer totalen Ordnung: Weglassen der $\rightarrow$

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**

$$
\begin{array}{lccccccc}
 & & & r_2[A] & \rightarrow & w_2[B] & \rightarrow & c_2 \\
 & & & \uparrow & & \uparrow & & \\
H = & r_1[A] & \rightarrow & w_1[A] & \rightarrow & w_1[B] & \rightarrow & c_1
\end{array}
$$

- Totale Ordnung

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

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

$H_1 \equiv H_2$ (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

$$
H = \begin{array}{ccccccc}
r_1[A] & \rightarrow & w_1[A] & \rightarrow & w_1[B] & \rightarrow & c_1 \\
 & & \uparrow & & \uparrow & & \\
 & & r_2[A] & \rightarrow & w_2[B] & \rightarrow & c_2 \\
 & & \downarrow & & & & \\
r_3[A] & \rightarrow & w_3[A] & \rightarrow & c_3 & &
\end{array}
$$

  - Zugehöriger Konfliktgraph

G(H):

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

  ➥ **Topologische Sortierung!**

- **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**

H =

$$w_1[A] \rightarrow w_1[B] \rightarrow c_1 \rightarrow r_2[A] \rightarrow r_3[B] \rightarrow w_2[A] \rightarrow c_2 \rightarrow w_3[B] \rightarrow c_3$$

- **Konfliktgraph**

G(H) :

- **Topologische Ordnungen**

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

$H_s^1 = T1 \mid T2 \mid T3$

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

$H_s^2 = T1 \mid T3 \mid T2$

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

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
      $w_j[A] \rightarrow ... \rightarrow r_i[A]$
  - Wie kritisch für das lokale Rücksetzen von T_j sind
    $r_i[A] \rightarrow ... \rightarrow w_j[A]$
    oder
    $w_j[A] \rightarrow ... \rightarrow w_i[A]$
    oder
    $w_i[A] \rightarrow ... \rightarrow w_j[A]$

- **Serialisierbarkeitstheorie:**
  Gebräuchliche Klassenbeziehungen[6]
  - SR: serialisierbare Historien
  - RC: rücksetzbare Historien
  - ACA: Historien ohne kaskadierendes Rücksetzen
  - ST: strikte Historien

6. 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**
  1. T_j schreibt mindestens ein Datum A, das T_i nachfolgend liest:
     $w_j[A] <_H r_i[A]$
  2. T_j wird (zumindest) nicht vor dem Lesevorgang von T_i zurückgesetzt:
     $a_j \not<_H r_i[A]$
  3. 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.

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

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

  $c_j <_H c_i$

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

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:

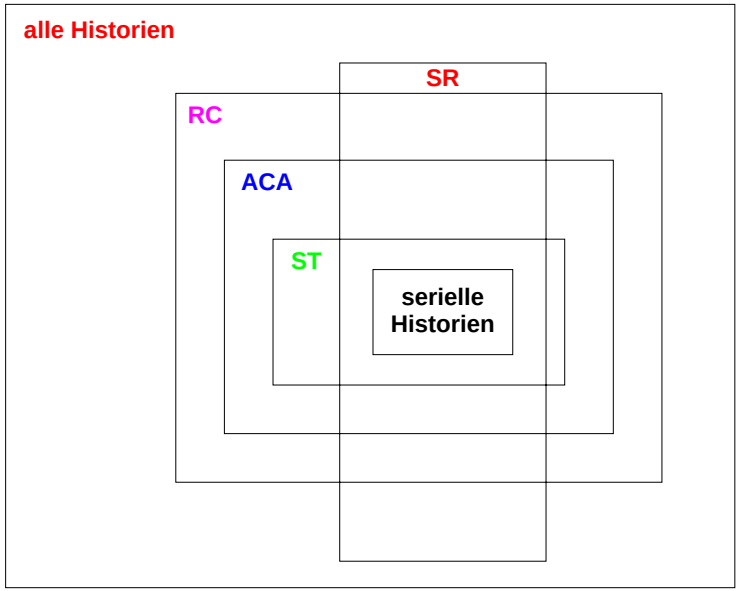
Wenn

$w_j[A] <_H o_i[A]$ (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**

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

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

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

angeforderter Modus der TA	aktueller Modus des Objekts: NL	R	X
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

- **Ablauf von Transaktionen**

T1	T2	a	b	Bem.
		NL	NL	
lock (a, X)		X		
. . .				
	lock (b, R)		R	
	. . .			
lock (b, R)			R	
	lock (a, R)	X		T2 wartet, WfG:
. . .				
unlock (a)		NL --> R		T2 wecken
. . .	. . .			
unlock(b)			R	

Zweiphasen-Sperrprotokolle[7]

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

- **Beispiel für ein 2PL-Protokoll** (2PL: two-phase locking)

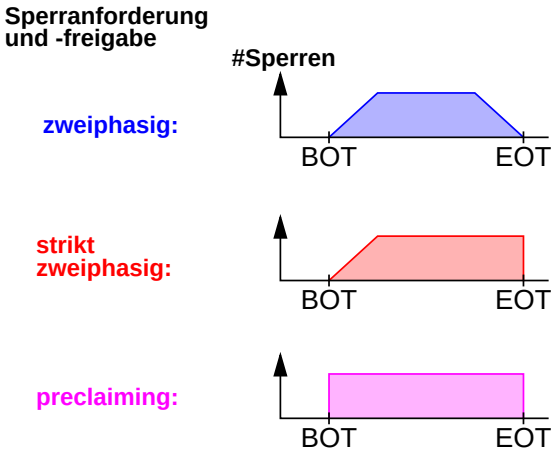
```
BOT
    lock (a, X)
    . . .
    lock (b, R)
    . . .
    lock (c, X)
    . . .
    unlock (b)
    unlock (c)
    unlock (a)
Commit
```

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

7. 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)

- **Formen der Zweiphasigkeit**



- **Anwendung des 2PL-Protokolls**

T1	T2	Bem.
BOT		
lock (a, X)		
read (a)		
write (a)		
	BOT	
	lock (a, X)	T2 wartet: WfG
lock (b, X)		
read (b)		
unlock (a)		T2 wecken
	read (a)	
	write (a)	
	unlock (a)	
	commit	
unlock (b)		

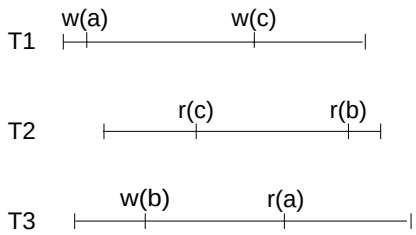
➥ **Zweiphasiges Protokoll** reicht für den praktischer Einsatz nicht aus!

Verklemmungen (Deadlocks)

- **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

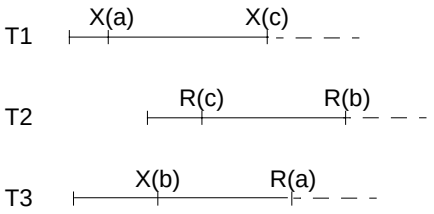
➥ Auftreten von Verklemmungen ist **inhärent** und kann bei pessimistischen Methoden (blockierende Verfahren) nicht vermieden werden.

- **Nicht-serialisierbare Historie**



SG

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



WfG

Logging und Recovery[8]

- **Aufgabe des DBMS:**
  Automatische Behandlung aller erwarteten Fehler

- **Was sind erwartete Fehler?[9]**
  - 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)
  - Bereitstellen von Wiederherstellungsalgorithmen im Fehlerfall (Recovery)

- **Logging**
  Sammeln von Redundanz im Normalbetrieb,
  um für den Fehlerfall gerüstet zu sein

---

8. Härder, T., Reuter, A.: Principles of Transaction Oriented Database Recovery, in: ACM Computing Surveys 15:4, Dec. 1983, 287-317.
9. Kommerzielle Anwendungen auf Großrechnern sind durch ihre Zuverlässigkeit gekennzeichnet. Nicht selten besteht der Code bis zu 90% aus (erprobten) Recovery-Routinen (W. G. Spruth).

Logging und Recovery (2)

- **„Recoverable actions“**
  - Zustand der aktuellen DB, die den DB-Pufferinhalt einschließt, und der der materialisierten DB (erreichbare DB-Daten auf Externspeicher) stimmen bei Crash nicht überein
  - ACID impliziert Robustheit, d. h., DB enthält nur solche Zustände, die explizit durch erfolgreich abgeschlossene TA erzeugt wurden
    - **D**auerhaftigkeit (Persistenz): Effekte von abgeschlossenen TA gehen nicht verloren
    - **A**tomarität (Resistenz): Zustandsänderungen werden entweder, wie in der TA spezifiziert, vollständig durchgeführt oder überhaupt nicht

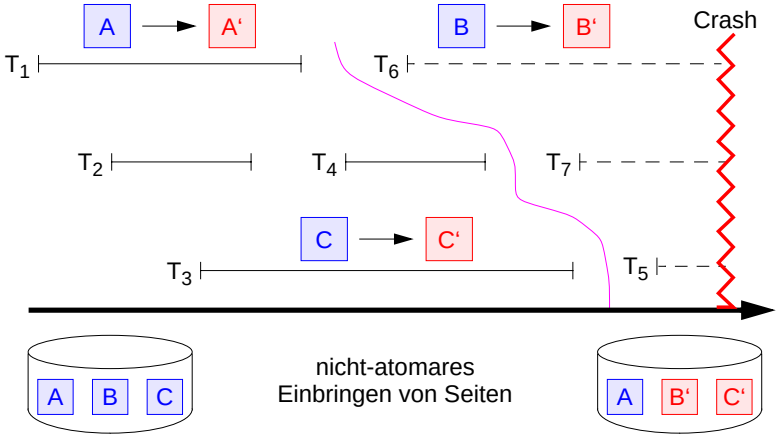
- **“A recoverable action is 30% harder and requires 20% more code than a non-recoverable action” (J. Gray)**
  - Zwei Prinzipien der Anweisungs-Atomarität möglich
    - **„Do things twice”**
      (vorbereitende Durchführung der Operation; wenn alles OK, erneuter Zugriff und Änderung)
    - **„Do things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

- **Zielzustand nach erfolgreicher Recovery:**

  Durch die Recovery-Aktionen ist der jüngste Zustand vor Erkennen des Fehlers wiederherzustellen, der allen semantischen Integritäts-bedingungen (Constraints des DB-Schemas) entspricht, der also ein möglichst aktuelles, exaktes Bild der Miniwelt darstellt

  ➥ ***jüngster transaktionskonsistenter DB-Zustand!***

Logging und Recovery (3)



- **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

- **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 Daten-bank (B‘ $\rightarrow$ B)

SQL-Operationshierarchie

Systemhierarchie + DB-Konsistenz im Fehlerfall

Log-Granulat

- TA-Programm
- DML.-Op.
- Eintrag
- Eintrag (elem.)
- Seite
- Archiv-Datei/ Archiv-Log

DB-Konsistenz und Logging (2)

- **DB-Konsistenz im Fehlerfall**
  - DB-Zustand bei Crash = materialisierte DB zum Zeitpunkt des Crashs
  - Eine bestimmte Konsistenz der materialisierten DB bedeutet, dass
    - die Effekte von Operationen der entsprechenden Abstraktionsebene vollständig in die DB eingebracht sind
    - keine Effekte von unvollständigen die DB erreicht haben
    - die Log-Informationen auf den DB-Zustand angewendet werden können
  - Wenn eine Einbringoperation beim Crash unterbrochen wurde, ist die Seite (der Block) i. Allg. unvollständig geschrieben!

  ➥ **Nur beim Seiten-Logging ruft eine unvollständig geschriebene Seite keine Medien-Recovery hervor!**

- **Auswahl eines Logging-Verfahrens**

  Wenn im Fehlerfall (Crash) die DB folgende Konsistenz aufweist:
  - Gerätekonsistenz → Seiten-Logging
  - Aktionskonsistenz (für Elementar-Ops) → Physiologisches Logging
  - Aktionskonsistenz (für interne Operationen) → Eintrags-Logging
  - Operationskonsistenz → DML-Op.-Logging (logisch, SQL-Ops)
  - TA-Konsistenz → TA-Programm-Logging (logisch)

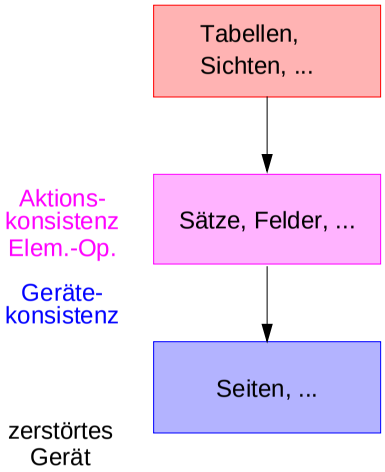
  ➥ **Der umgekehrte Schluss ist nicht zwingend!**

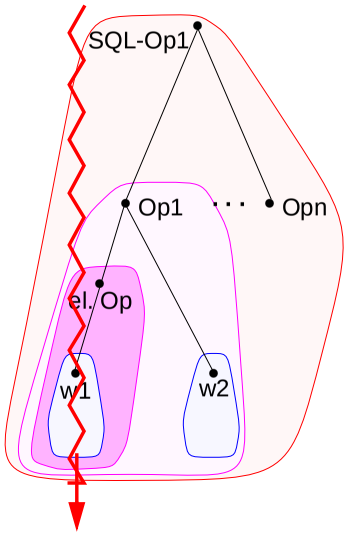
- **Garantie einer bestimmten Konsistenz**
  - Wenn bei Crash die Konsistenz einer Abstraktionsebene garantiert wird, können **Logging-Verfahren niedrigerer Konsistenzebene** gewählt werden
  - Dieser Fall tritt üblicherweise nicht auf, da die Gewährleistungskosten für die Konsistenz mit der Abstraktionsebene steigen!

Nicht-atomare Einbringverfahren

- **Allgemeine Eigenschaft: DB ist beim Crash „chaos-konsistent“**

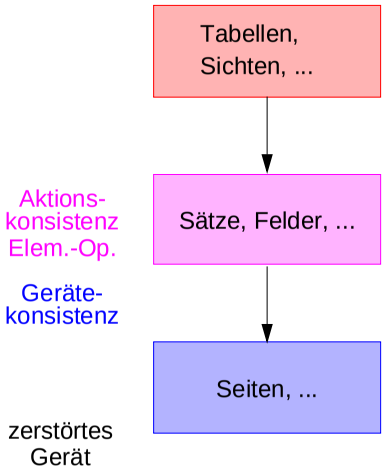
- **Wenn Seiten zerstört sind, ist i. Allg. Medien-Recovery erforderlich**

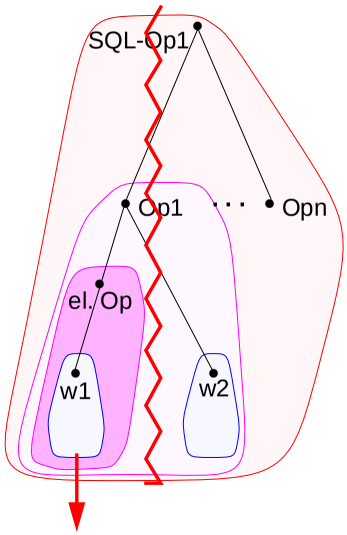




Nicht-atomare Einbringverfahren (2)

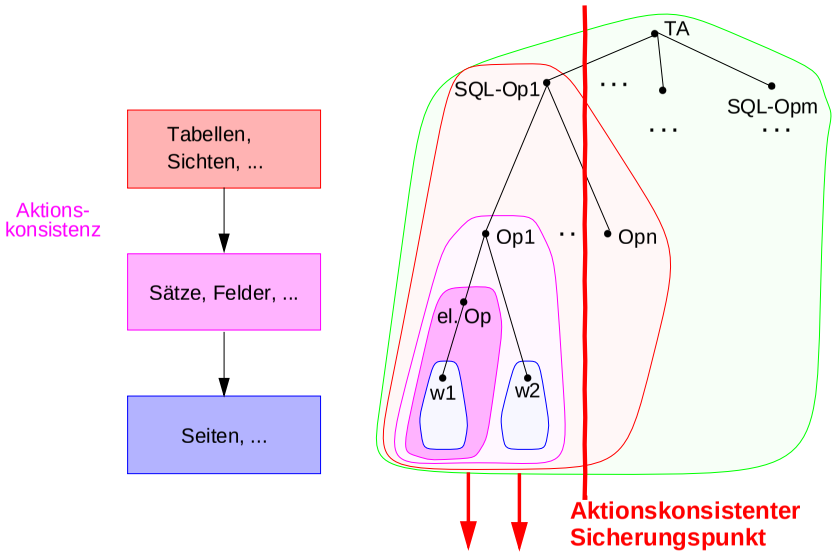
- **Wenn Gerätekonsistenz als Minimalbedingung vorliegt:**
  - bei Seiten-Logging: Austausch ganzer Seiten (extrem teuer!)
  - Trick: Physiologisches Logging mit LSNs
    (physical to a page, logical within a page)





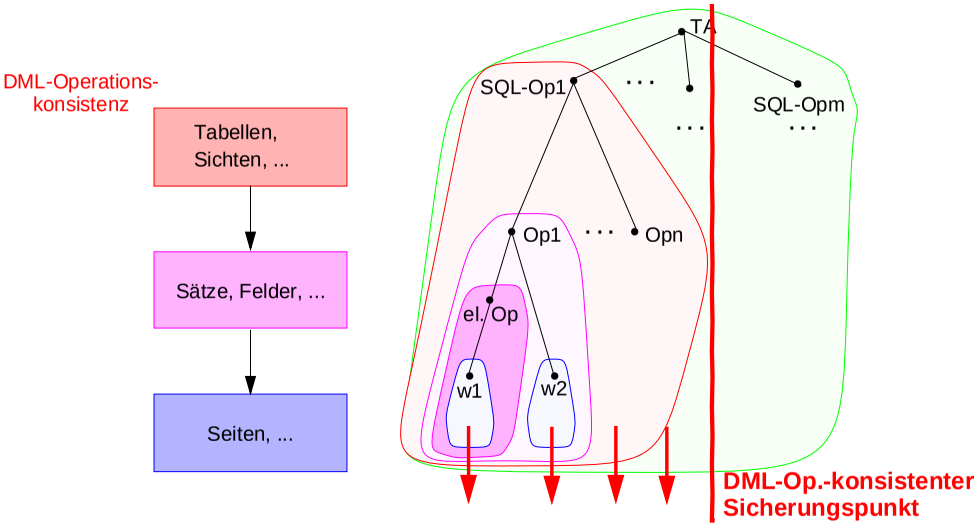
Atomare Einbringverfahren

- **Operationen höherer Schichten betreffen mehrere Seiten**
  - Log-Einheit muss Undo/Redo in mehreren Seiten durchführen
  - Betroffene Seitenmenge muss **vollständig oder überhaupt nicht** in der DB sein
- **Rolle von Sicherungspunkten: Begrenzung der Redo-Recovery beim Crash**



Atomare Einbringverfahren (2)

- **Sicherungspunkte bei atomaren Einbringstrategien:**
  Zustand der materialisierten DB bleibt bis zum nächsten erfolgreichen SP erhalten
- **Logisches Logging: DML-Operationen**
  Verlorengegangene Änderungen von DML-Operationen können wiederholt werden;
  die DML-Operationen können beim Crash auf der materialisierten DB ausgeführt werden



Atomare Einbringverfahren (3)

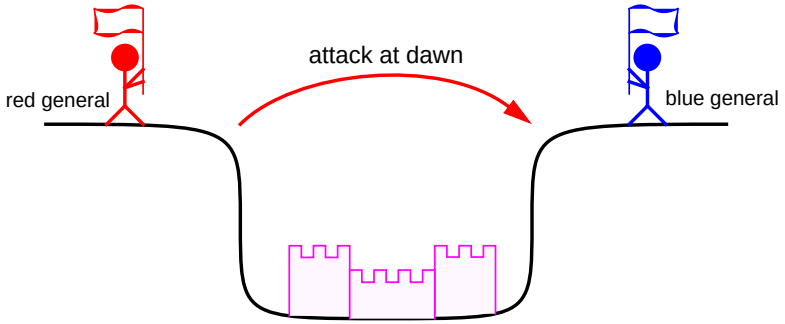
- **Transaktionskonsistente Sicherungspunkte:**
  Materialisierte DB ist stets transaktionskonsistent. Erhaltung dieser Eigenschaft ist aber sehr teuer
- **Transaktions-Logging: Im Prinzip genügen Eingabeparameter des Transaktionsprogramms**
  Verlorengegangene Änderungen von erfolgreichen Transaktionen müssen beim Crash in Serialisierbarkeitsreihenfolge im Einbenutzerbetrieb ausgeführt werden

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**



Generals-Paradoxon

- **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**

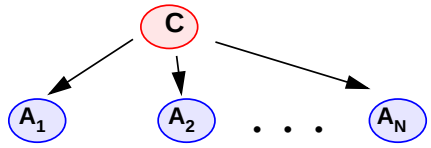
Verarbeitung in Verteilten Systemen (2)

- **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*

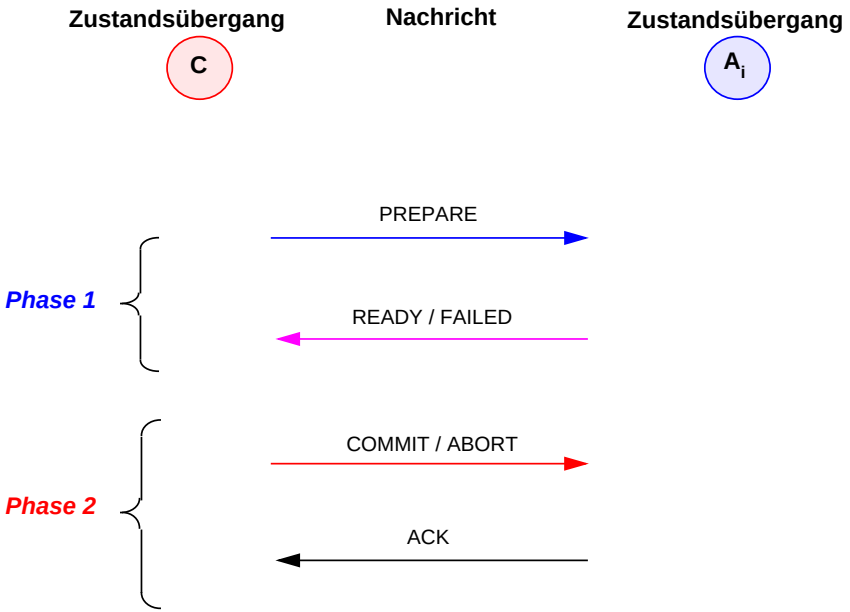
- **Anforderungen an geeignetes Commit-Protokoll:**
  - Geringer Aufwand (#Nachrichten, #Log-Ausgaben)
  - Minimale Antwortzeitverlängerung (Nutzung von Parallelität)
  - Robustheit gegenüber Rechnerausfällen und Kommunikationsfehlern

➥ 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



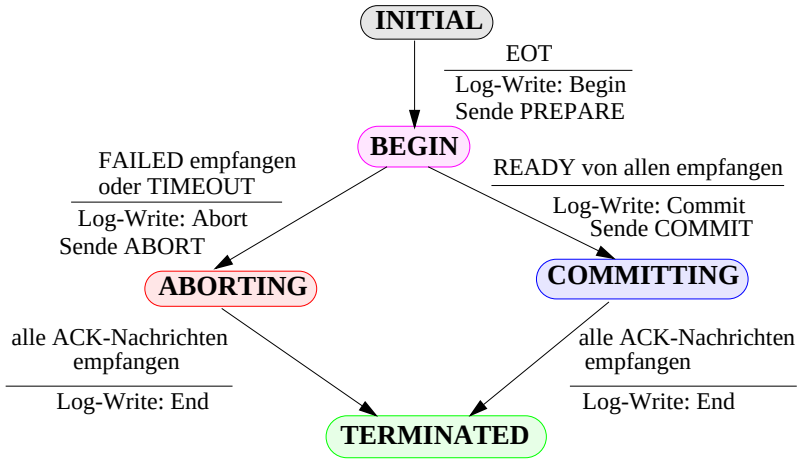
- **Protokoll erfordert Folge von Zustandsübergängen**
  - für Koordinator
  - für jeden Agenten

➥ Zustandsübergänge müssen auf „sicherem Platz“ (Log) vermerkt sein!
(Übergang nach TERMINATED braucht nicht synchron zu erfolgen)

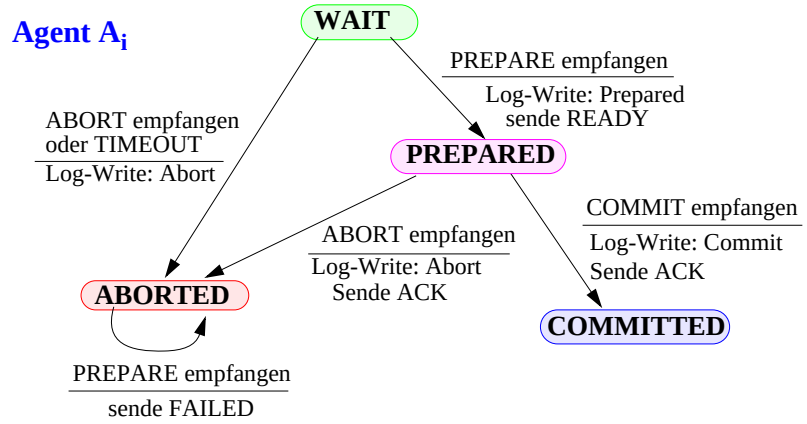
- **Aufwand im Erfolgsfall:**
  - Nachrichten:
  - Log-Ausgaben (forced log writes):

2PC: Zustandsübergänge

Koordinator C



Agent A_i



2PC: Fehlerbehandlung

- **Timeout-Bedingungen für Koordinator:**
  - **BEGIN** ➥ setze Transaktion zurück; verschicke ABORT-Nachricht
  - **ABORTING**, **COMMITTING** ➥ vermerke Agenten, für die ACK noch aussteht
- **Timeout-Bedingungen für Agenten:**
  - **WAIT** ➥ setze Teiltransaktion zurück (unilateral ABORT)
  - **PREPARED** ➥ erfrage Transaktionsausgang bei Koordinator (bzw. anderen Rechnern)
- **Ausfall des Koordinatorknotens:**
  Vermerkter Zustand auf Log
  - **TERMINATED:**
    - UNDO bzw. REDO-Recovery, je nach Transaktionsausgang
    - keine "offene" Teiltransaktionen möglich
  - **ABORTING:**
    - UNDO-Recovery
    - ABORT-Nachricht an Rechner, von denen ACK noch aussteht
  - **COMMITTING:**
    - REDO-Recovery
    - COMMIT-Nachricht an Rechner, von denen ACK noch aussteht
  - Sonst: UNDO-Recovery
- **Rechnerausfall für Agenten:**
  Vermerkter Zustand auf Log
  - **COMMITTED:** REDO-Recovery
  - **ABORTED** bzw. kein 2PC-Log-Satz vorhanden: UNDO-Recovery
  - **PREPARED:** Anfrage an Koordinator-Knoten, wie TA beendet wurde (Koordinator hält Information, da noch kein ACK erfolgte)

Commit: Kostenbetrachtungen

- **vollständiges 2PC-Protokoll**
  (N = #Teil-TA, davon M = #Leser)
  - Nachrichten: 4 N
  - Log-Ausgaben: 2 + 2 N
  - 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:
  - Log-Ausgaben:
    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
- 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
- **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)

    ➥ **Sperrverfahren sind pessimistisch und universell einsetzbar**

Zusammenfassung (2)

- **Fehlerarten: Transaktions-, System-, Gerätefehler und Katastrophen**
- **Breites Spektrum von Logging- und Recovery-Verfahren**
  - Logging kann auf verschiedenen Systemebenen angesiedelt werden
  - erfordert schichtenspezifische Konsistenz im Fehlerfall
- **Synchronisationsgranulat muss größer oder gleich dem Log-Granulat sein**
- **Atomare Einbringverfahren**
  - erhalten den DB-Zustand des letzten Sicherungspunktes
  - gewährleisten demnach die gewählte schichtenspezifische DB-Konsistenz auch bei der Recovery von einem Crash und
  - erlauben folglich Eintrags-, DML- oder Transaktions-Logging
- **Nicht-atomare Einbringverfahren**
  - sind i. Allg. atomaren Einbringverfahren vorzuziehen, weil sie im Normalbetrieb wesentlich billiger sind und
  - nur eine geringe Crash-Wahrscheinlichkeit zu unterstellen ist
  - Sie erfordern jedoch Seiten-Logging oder physiologisches Logging
- **Zweiphasen-Commit-Protokolle**
  - Hoher Aufwand an Kommunikation und E/A
  - Optimierungsmöglichkeiten sind zu nutzen
  - Massnahmen erforderlich, um Blockierungen zu vermeiden!

    ➥ **Kritische Stelle:** Ausfall von C