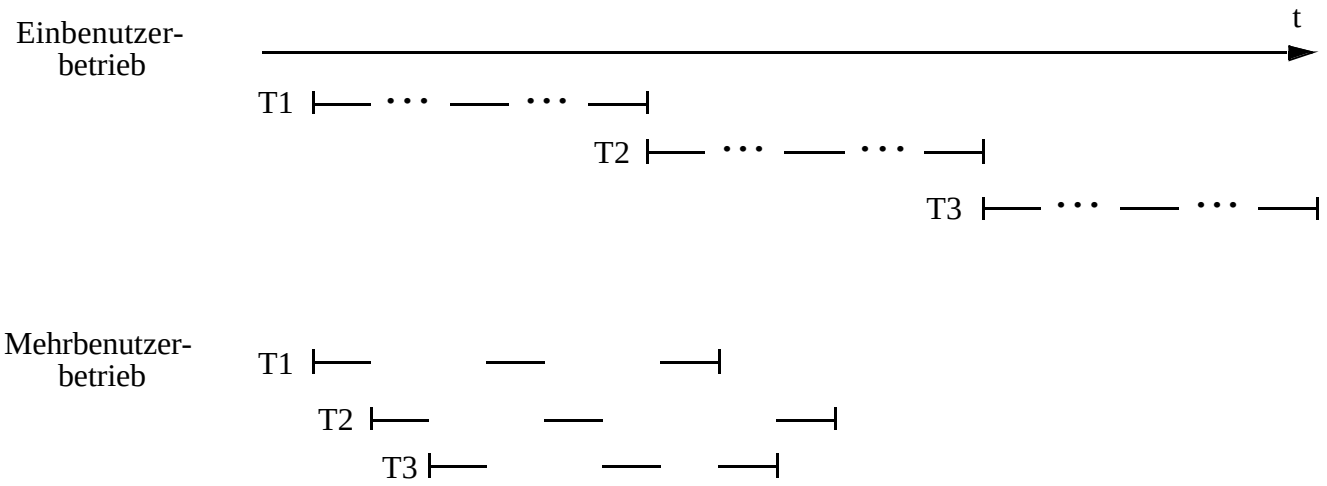
6. Serialisierbarkeit[1]

- **Anomalien im Mehrbenutzerbetrieb**
  - Verlorengegangene Änderungen
  - Inkonsistente Analyse, Phantom-Problem usw.
- **Synchronisation von Transaktionen**
  - Ablaufpläne, Modellannahmen
  - Korrektheitskriterium
  - Konsistenzerhaltende Ablaufpläne
- **Theorie der Serialisierbarkeit**
  - Beschränkung auf Konfliktserialisierbarkeit
  - Äquivalenz von Historien
  - Serialisierbarkeitstheorem
  - Klassen von Historien
    - SR: serialisierbare Historien
    - RC: rücksetzbare Historien
    - ACA: Historien ohne kaskadierendes Rücksetzen
    - ST: strikte Historien
    - S: serielle Historien
- **Beispiele**

1. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems, Addison-Wesley Publ. Comp., 1987 (http://research.microsoft.com/pubs/ccontrol/)

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 Cursorn

å **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:

Lesetransaktion (Gehaltssumme berechnen)	Änderungstransaktion	DB-Inhalt (Pnr, Gehalt)
		2345 39.000
SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 2345		3456 48.000
summe := summe + gehalt		
	UPDATE Pers SET Gehalt = Gehalt + 1000 WHERE Pnr = 2345	
		2345 40.000
	UPDATE Pers SET Gehalt = Gehalt + 2000 WHERE Pnr = 3456	
		3456 50.000
SELECT Gehalt INTO :gehalt FROM Pers WHERE Pnr = 3456		
summe := summe + gehalt		

Zeit

Phantom-Problem

Einfügungen oder Löschungen können Leser zu falschen Schlußfolgerungen verleiten:

Lesetransaktion (Gehaltssumme überprüfen)	Änderungstransaktion (Einfügen eines neuen Angestellten)
SELECT SUM (Gehalt) INTO :summe FROM Pers WHERE Anr = 17	
	INSERT INTO Pers (Pnr, Anr, Gehalt) VALUES (4567, 17, 55.000)
	UPDATE Abt SET Gehaltssumme = Gehaltssumme + 55.000 WHERE Anr = 17
SELECT Gehaltssumme INTO :gsumme FROM Abt WHERE Anr = 17	
IF gsumme <> summe THEN <Fehlerbehandlung>	

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);			

  å **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	

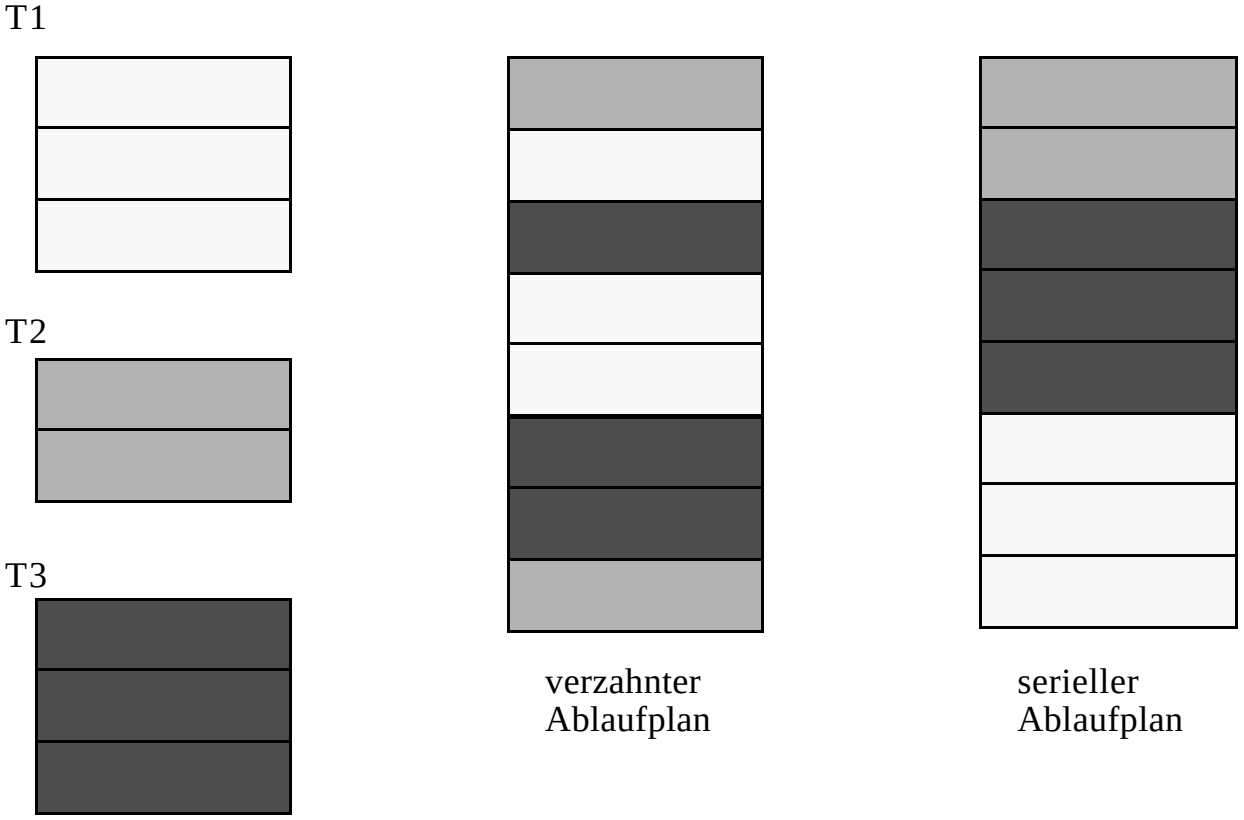
  å **Cursor-Stabilität sollte gewährleistet werden!**

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**



å Wenn Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten.

- **Ziel der Synchronisation:**

  logischer Einbenutzerbetrieb,
  d.h. Vermeidung aller Mehrbenutzeranomalien

  å **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)

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

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

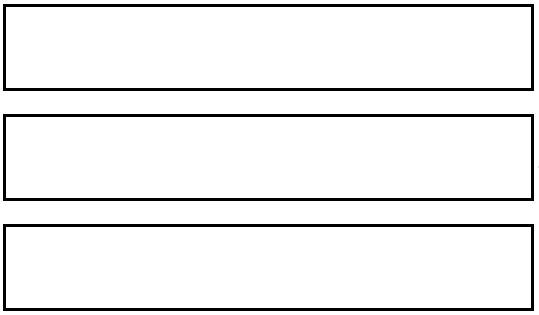
	A	B	C	A + B + C
initialer Wert				
nach T1; T2				
nach Ausf. 2				
nach Ausf. 3				
nach T2; T1				

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

Synchronisation - Modellannahmen

- **Modellbildung**

  für die Synchronisation



- **Read/Write-Modell (Page Model)**
  - DB ist Menge von unteilbaren, uninterpretierten Datenobjekten (z. B. Seiten)
  - DB-Anweisungen lassen sich nachbilden durch atomare Lese- und Schreib-operationen 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:

    $T = p_1\, p_2\, p_3 \ldots p_n$ 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

    $T = p_1 \ldots p_n\, a$ oder $T = p_1 \ldots p_n\, c$

  å **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 kann durch einen *Historie (Schedule*) beschrieben werden:**

  Beispiel:

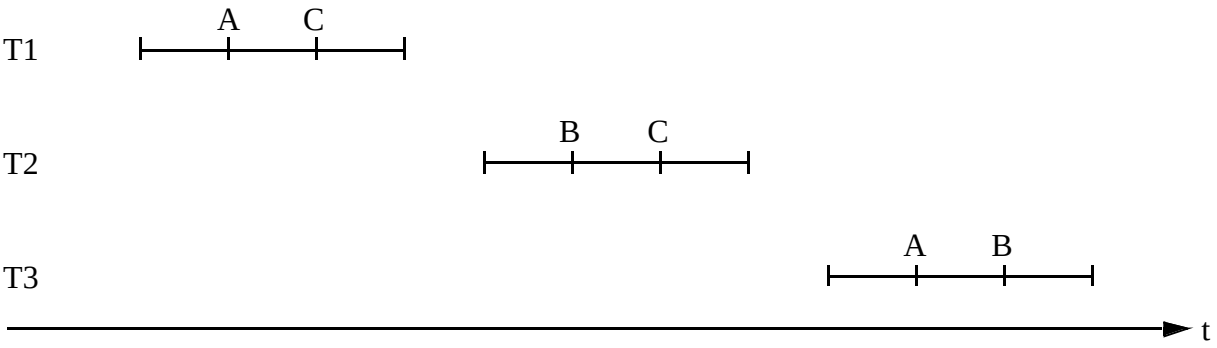
  $r_1[A]\ r_2[A]\ r_3[B]\ w_1[A]\ w_3[B]\ r_1[B]\ c_1\ r_3[A]\ w_2[A]\ a_2\ w_3[C]\ c_3$ ...

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

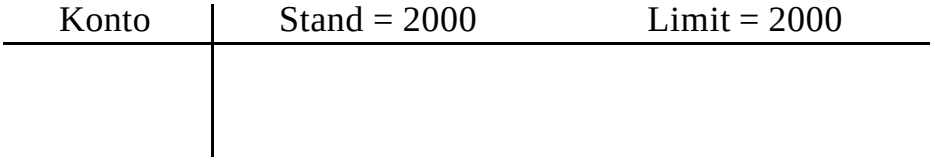
- Die TA T1-T3 müssen so synchronisiert werden, daß der resultierende Zustand der DB gleich dem ist, der bei der seriellen Ausführung in einer der folgenden Sequenzen zustande gekommen wäre:

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

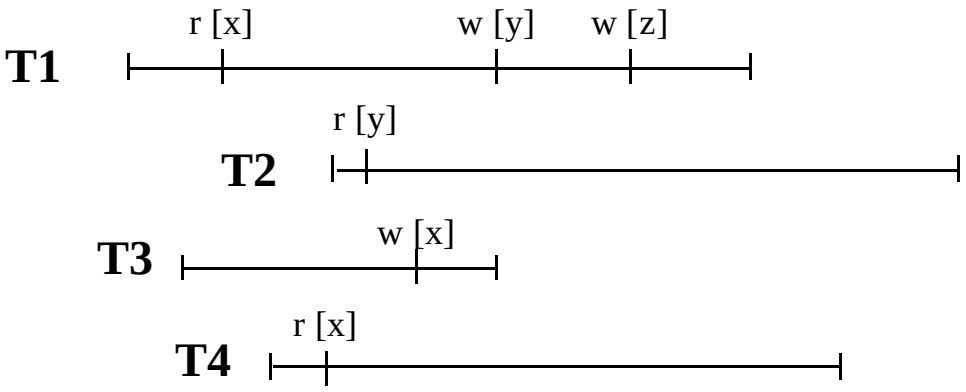
- Bei n TA gibt es n! (hier 3! = 6) mögliche serielle Ablaufpläne

- **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!

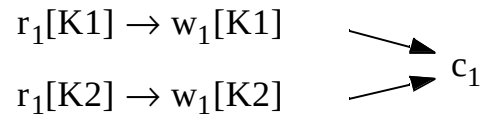
- **Konsistenzanforderungen an eine TA**
  - Falls T_i ein **abort** durchführt, müssen alle anderen Operationen $p_i[A]$ vor a_i ausgeführt werden: $p_i[A] <_i a_i$
  - Analoges gilt für das **commit:** $p_i[A] <_i c_i$
  - Wenn T_i ein Datum A liest und auch schreibt, ist die **Reihenfolge festzulegen**:

    $r_i[A] <_i w_i[A]$ 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**[1]
  - 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 å 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

    å **kein Konflikt!**

  - $r_i[A]$ und $w_j[A]$: Reihenfolge ist relevant und festzulegen.

    Entweder $r_i[A) \rightarrow w_j[A]$

    å **R/W-Konflikt!**

    oder $w_j[A] \rightarrow r_i[A]$

    å **W/R-Konflikt!**

  - $w_i[A]$ und $r_j[A]$: analog

  - $w_i[A]$ und $w_j[A]$ Reihenfolge ist relevant und festzulegen
    å **W/W-Konflikt!**

1. 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**[1]

- **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

1. 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**

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

- Reihenfolge konfliktfreier Operationen (zwischen TA) wird nicht spezifiziert

- Mögliche totale Ordnung[1]

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

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

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

Theorie der Serialisierbarkeit (5)

- **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**

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

  - 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 Serialisierungsgraphen 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

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

  - Zugehöriger Konfliktgraphgraph

    G(H):

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

  å **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[1]
  - SR: serialisierbare Historien
  - RC: rücksetzbare Historien
  - ACA: Historien ohne kaskadierendes Rücksetzen
  - ST: strikte Historien

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

Rücksetzbare Historie

- **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.

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

- **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**)				

å **In der Theorie ist ACID garantierbar! Aber . . .**

- **Eine Historie vermeidet kaskadierendes Rücksetzen,** wenn

$$c_j <_H r_i[A]$$

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

å **Änderungen dürfen erst nach Commit freigegeben werden**

Klassen von Historien

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

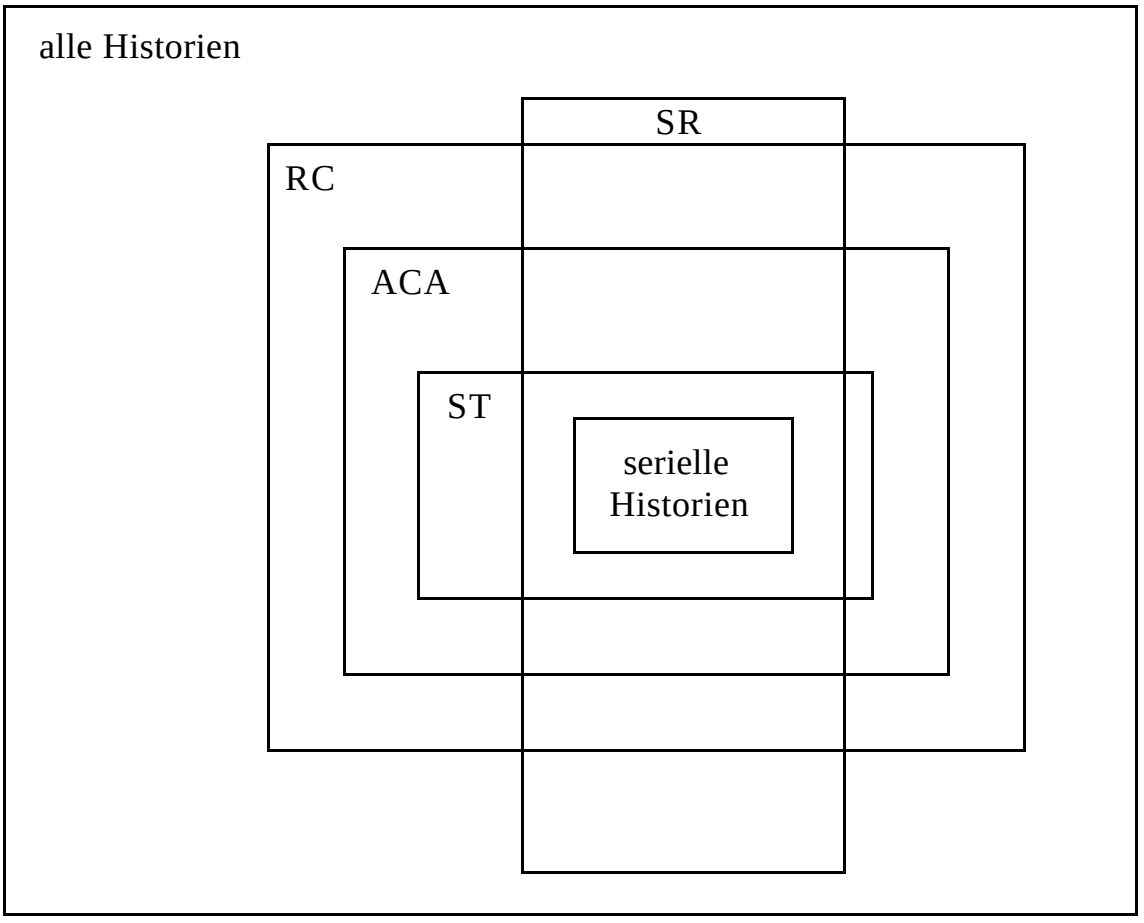
  Wenn

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



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

Zusammenfassung

- Beim ungeschützten und konkurrierenden Zugriff von Lesern und Schreibern auf gemeinsame Daten können Anomalien auftreten

- **Korrektheitskriterium der Synchronisation: Serialisierbarkeit**
  (gleicher DB-Zustand, gleiche Ausgabewerte wie bei seriellem Ablaufplan)

- **Theorie der Serialisierbarkeit**
  - einfaches Read/Write-Modell (Syntaktische Behandlung)
  - **Konfliktoperationen**:
    Kritisch sind Operationen verschiedener Transaktionen auf **denselben DB-Daten,** wenn diese Operationen **nicht reihenfolgeunabhängig** sind!
  - Im Gegensatz zur Final-State-Serialisierbarkeit (Klasse FSR) und View-Serialisierbarkeit (Klasse VSR) ist **Konflikt-Serialisierbarkeit** (Klasse CSR) für praktische Anwendungen die wichtigste. Sie ist effizient überprüfbar und unterscheidet sich bereits dadurch wesentlich von den beiden anderen Serialisierbarkeitsbegriffen.
    Es gilt: $CSR \subset VSR \subset FSR$
  - **Serialisierbarkeitstheorem**:
    Eine Historie H ist genau dann serialisierbar, wenn der zugehörige Konfliktgraph G(H) azyklisch ist
  - weitergehende Ansätze: Einbezug der Anwendungssemantik
    (Synchronisation von abstrakten Operationen auf Objekten)
  - enorm gründlich erforscht

- **Serialisierbare Abläufe**
  - gewährleisten „automatisch“ Korrektheit des Mehrbenutzerbetriebs
  - Anzahl der möglichen Historien (Schedules) bestimmt erreichbaren Grad an Parallelität