5. Synchronisation[1]

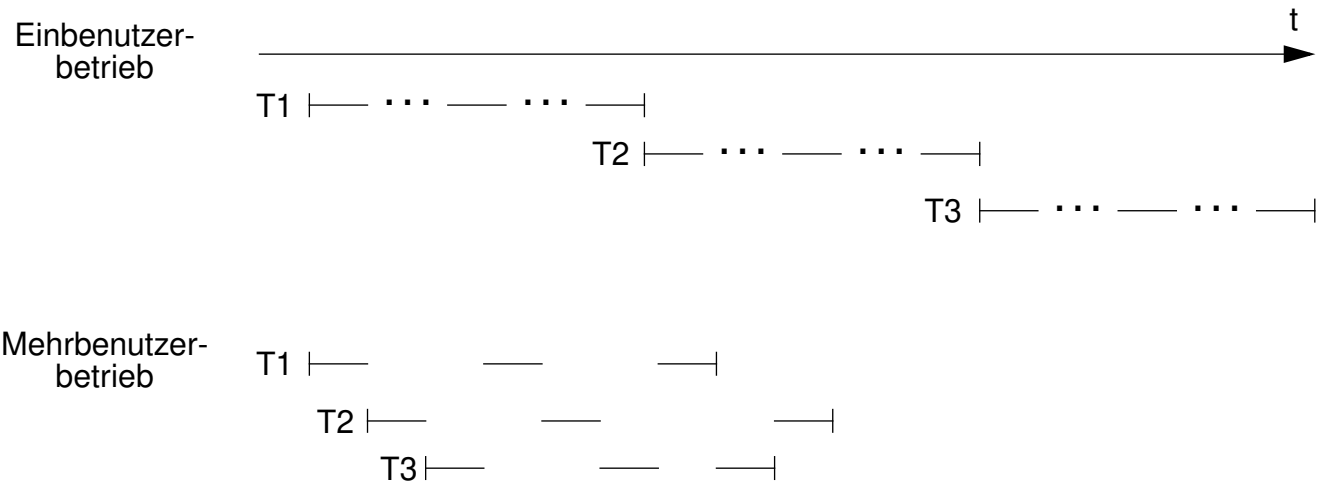
- **Anomalien im Mehrbenutzerbetrieb**
  - Abhängigkeit von nicht freigegebenen Änderungen
  - Verlorengegangene Änderung, Inkonsistente Analyse, Phantome, ...
- **Synchronisation von Transaktionen**
  - Ablaufpläne, Modellannahmen
  - Korrektheitskriterium
- **Theorie der Serialisierbarkeit**
  - Serialisierbare Historien
  - Klassen von Historien
- **Wie wird Synchronisation implementiert?**
- **Zweiphasen-Sperrprotokolle**
  - RX-Protokoll, RUX-Protokoll
  - Hierarchische Sperrverfahren
    - Anwartschaftssperren: IR, IX
    - RIX-Modus
- **Optimistische Synchronisation**
  - Eigenschaften
  - BOCC, FOCC
- **Deadlock-Behandlung**
- **Konsistenzebenen**
  - Inkaufnahme von Anomalien vs. Reduktion der Parallelität
  - (teilweise) Kontrolle der Programmierers vs. vollständige Systemkontrolle

---

1. Thomasian, A.: Concurrency Control: Methods, Performance, and Analysis, in: ACM Computing Surveys 30:1, March 1998, pp. 70-119.

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

➥ **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)

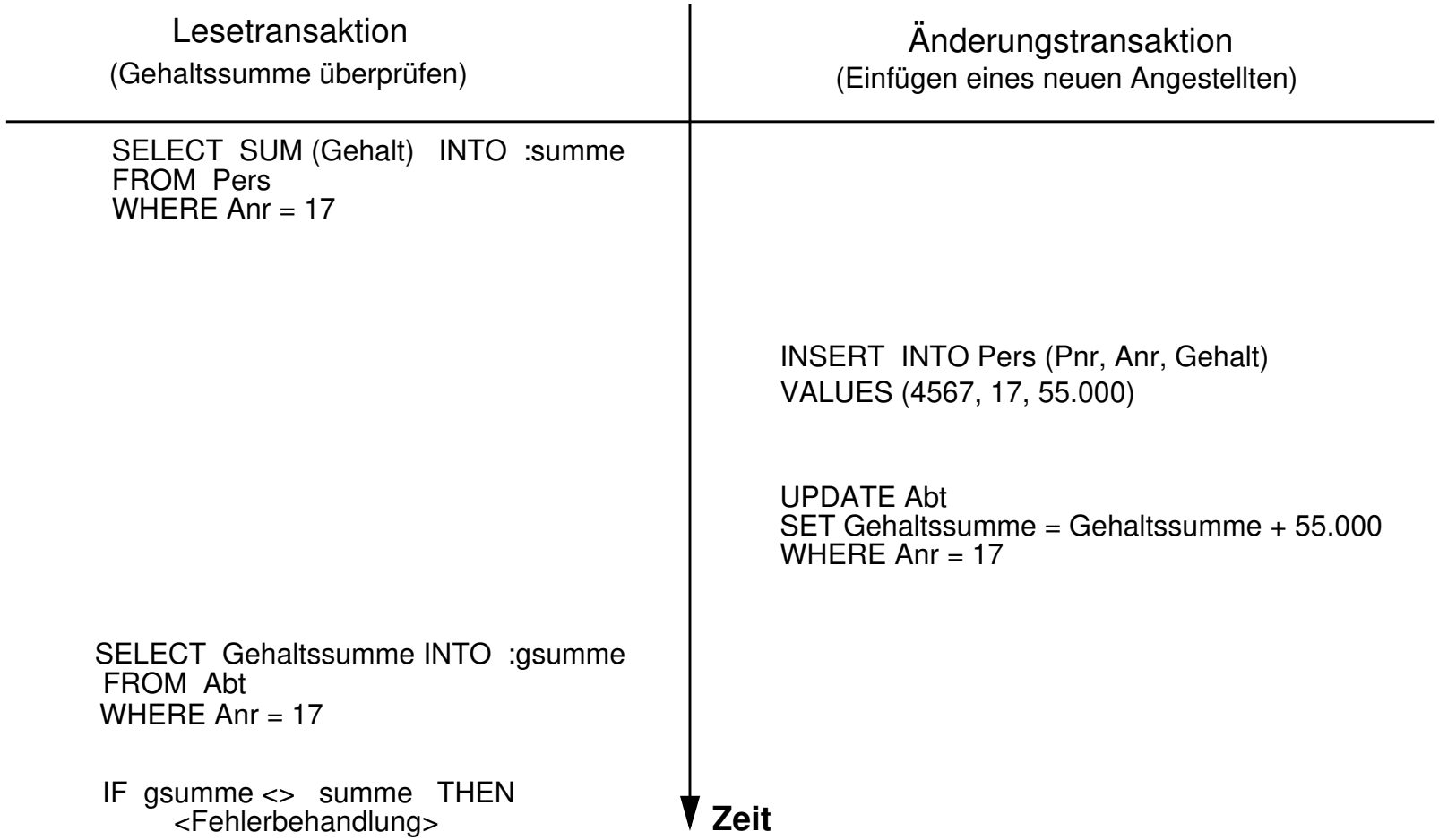
Das wiederholte Lesen einer gegebenen Folge von Daten führt auf verschiedene Ergebnisse:

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:



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	

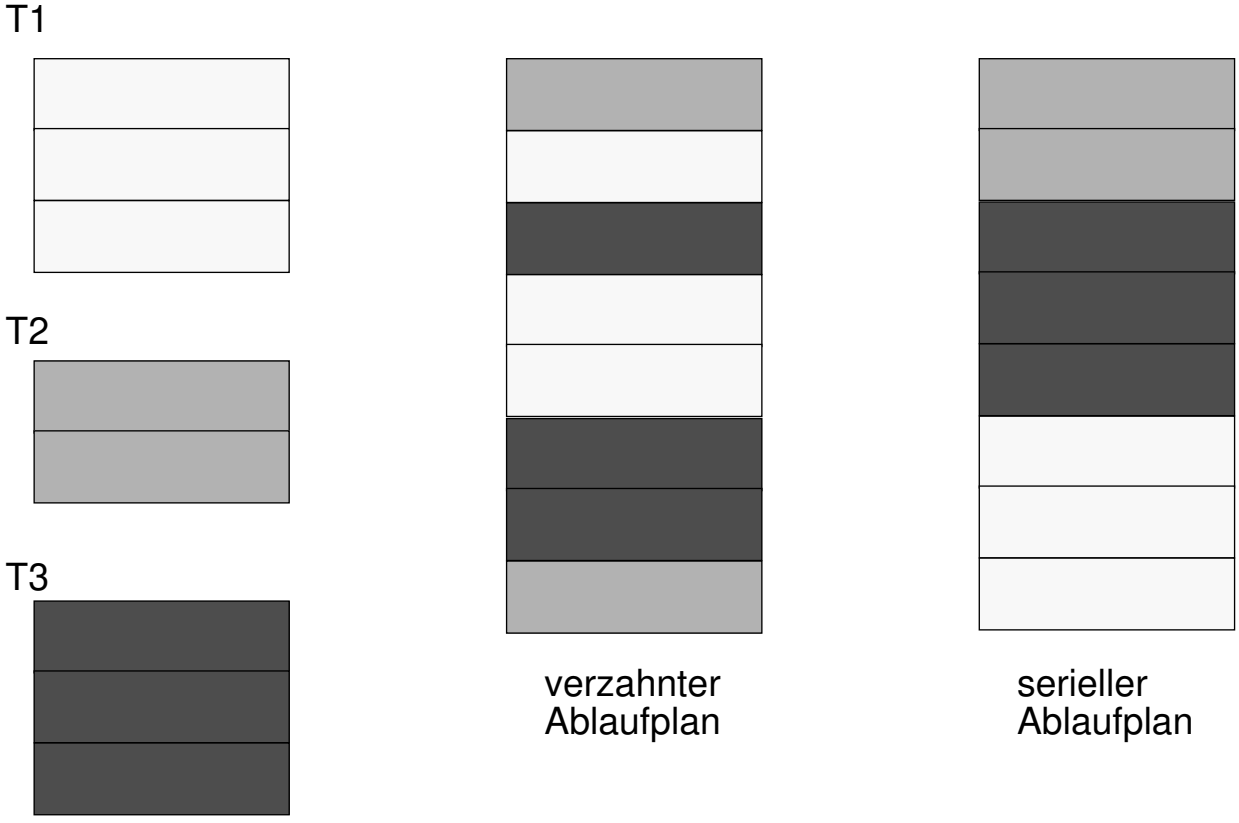
  ➥ **Cursor-Stabilität sollte gewährleistet werden!**

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 werden keine Konsistenzgarantien eingehalten)

- **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

  ➥ **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				

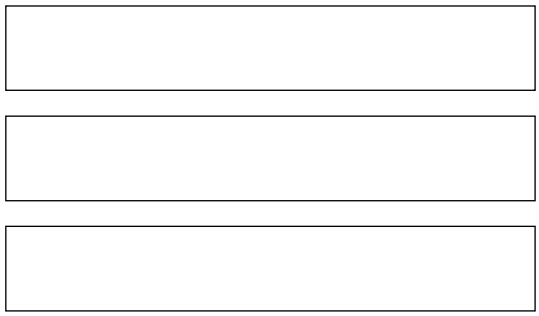
- **Ziel:** Äquivalenz der Ergebnisse von verzahnten Ausführungen zu einer der möglichen seriellen Ausführungen

➥ **Serielle Ausführungen können verschiedene Ergebnisse haben!**

Synchronisation - Modellannahmen

- **Modellbildung**

  für die Synchronisation



- **Read/Write-Modell**
  - 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:

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

  ➥ **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 *Schedule* (Ablaufplan) beschrieben werden:**

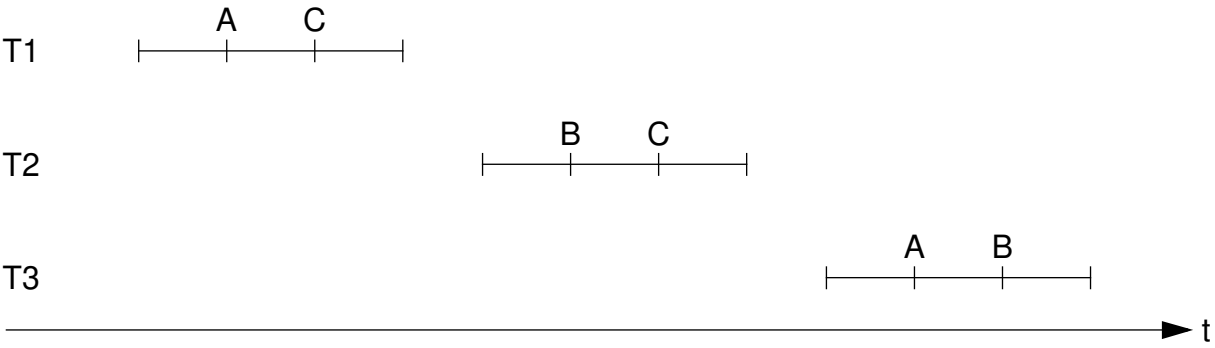
  Beispiel:
  $r_1(A), r_2(A), r_3(B), w_1(A), w_3(B), r_1(B), c_1, r_3(A), w_2(A), a_2, w_3(C), c_3, \ldots$

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

Theorie der Serialisierbarkeit[1]

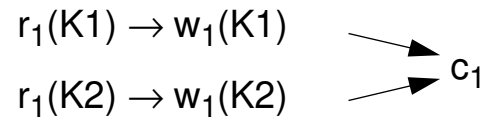
- **Ablauf einer Transaktion**
  - Häufigste Annahme: streng sequentielle Reihenfolge der Operationen
  - Serialisierbarkeitstheorie läßt sich auch auf der Basis einer partiellen Ordnung ($<_i$) entwickeln
  - TA-Abschluß: **abort** oder **commit** - aber nicht beides!
- **Konsistenzanforderungen an 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



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/)

Theorie der Serialisierbarkeit (2)

- **Historie (Schedule)**
  - 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!**

Theorie der Serialisierbarkeit (3)

- **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

     $q <_H p$

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]

1. Alternative Schreibweise bei einer totalen Ordnung: Weglassen der →

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 Operation **innerhalb** einer TA bleibt **invariant**

- **Beispiel**

$$
\begin{array}{llllll}
 & & r_2(A) & \rightarrow & w_2(B) & \rightarrow & c_2 \\
 & & \uparrow & & \uparrow & & \\
H = & r_1(A) & \rightarrow & w_1(A) & \rightarrow & w_1(B) & \rightarrow & c_1
\end{array}
$$

  - Totale Ordnung

    $H_1 =$

Serialisierbare Historie

- **Eine Historie H ist serialisierbar, wenn sie äquivalent zu einer seriellen Historie H_s ist**

  - Einführung eines Serialisierbarkeitsgraphen SG(H)
    - Konstruktion des SG(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 SG(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 Serialisierbarkeitsgraph

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

  ➥ **Topologische Sortierung!**

Serialisierbare Historie (2)

- **Historie**

  H =

  $w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$

- **Serialisierbarkeitsgraph**

- **Topologische Ordnungen**

  $H_s^1 = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow$

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

  $H_s^2 = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow$

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

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

Serialisierbare Historie (3)

- **Anforderungen an im DBMS zugelassene Historien (Schedules)**
  - 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
      $w_j(A) \rightarrow ... \rightarrow r_i(A)$

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

1. Bernstein, P.A., Hadizilacos, V.; Goodman, N.: Concurreny Control and Recovery in Database Systems, Addison-Wesley, 1987

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.

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

  $c_j <_H c_i$

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 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.

➥ **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:

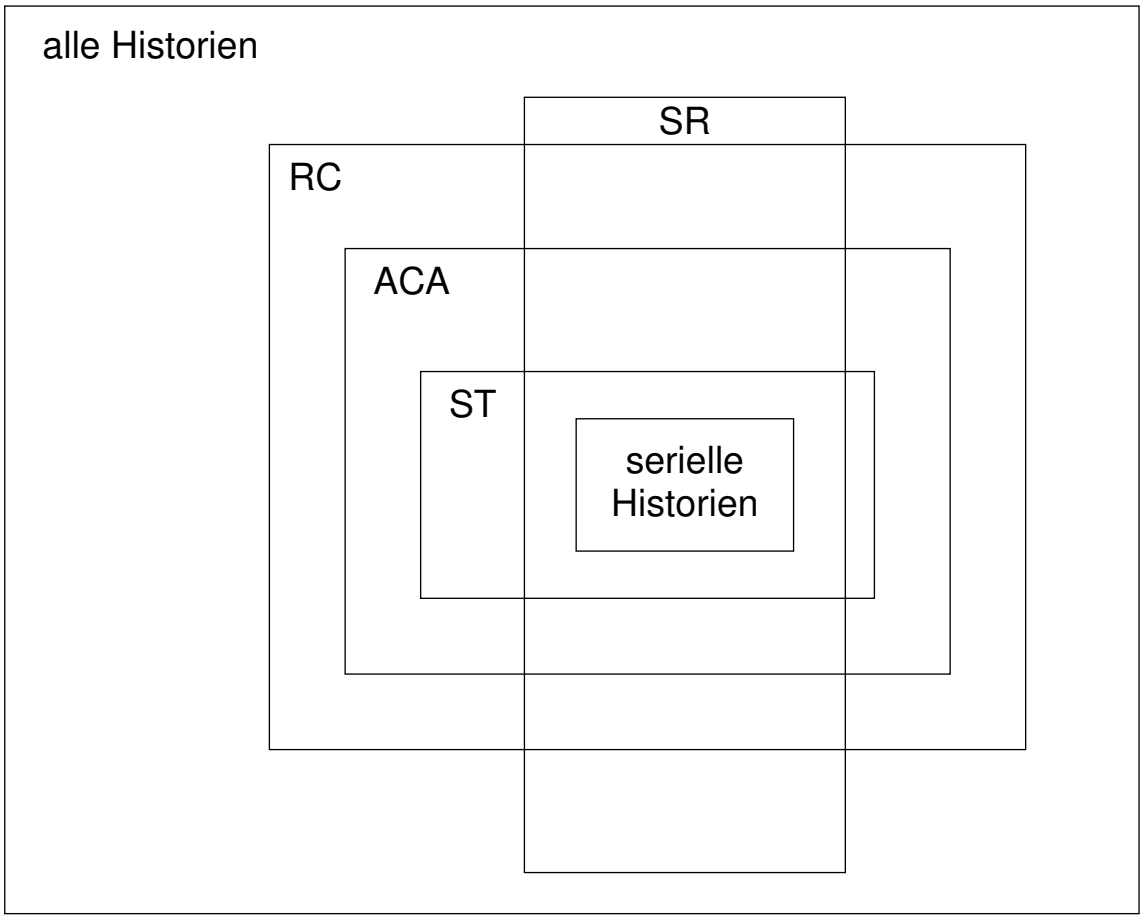
  Wenn

  $w_j(A) <_H o_i(A)$ (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**



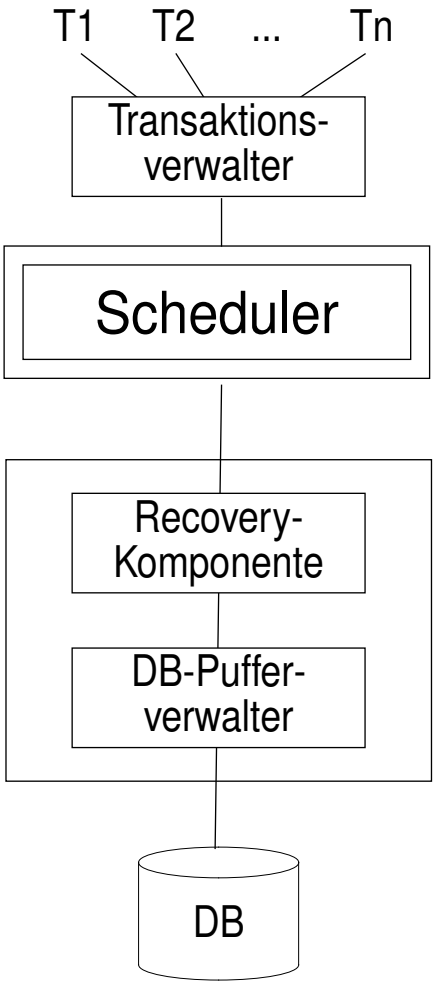
➥ **Schlußfolgerungen?**

Klassen von Historien (2)

- **Beispiele**

Einbettung des DB-Schedulers

- **Stark vereinfachtes Modell**



- **Welche Aufgaben hat der Scheduler?**

Synchronisationsverfahren

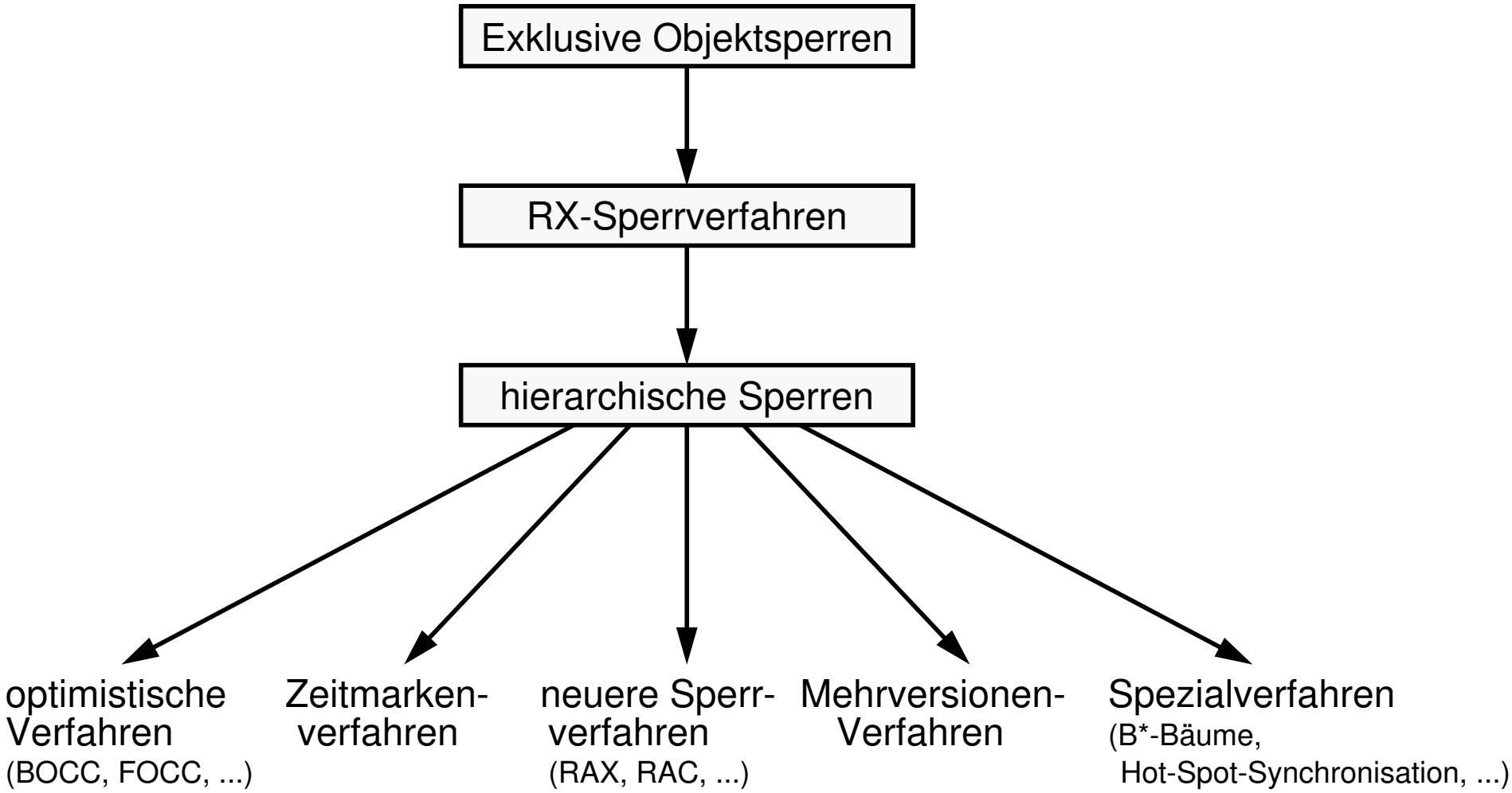
- **Scheduler**
  - Komponente der Transaktionsverwaltung
  - kontrolliert die beim TA-Ablauf auftretenden Konfliktoperationen (R/W, W/R, W/W) und garantiert insbesondere, daß nur „serialisierbare“ TA erfolgreich beendet werden
  - Nicht serialisierbare TA müssen verhindert werden.
    Dazu ist eine Kooperation mit der Recovery-Komponente erforderlich (Rücksetzen von TA)

- **Zur Realisierung der Synchronisation gibt es viele Verfahren**
  - Pessimistische Verfahren: Bei einer Konfliktoperation blockieren sie den Zugriff auf das Objekt.
  - Optimistische Verfahren: Erst bei Commit wird überprüft, ob die TA serialisierbar ist
  - Versionsverfahren, Zeitmarkenverfahren usw.
  - . . .

  ➥ Sperrverfahren sind pessimistisch und universell einsetzbar.

- **Sperrbasierte Synchronisation**
  - Sperren stellen während des laufenden Betriebs sicher, daß die resultierende Historie serialisierbar bleibt
  - Es gibt mehrere Varianten

Historische Entwicklung von Synchronisationsverfahren



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 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
. . .				
unlock (a)		NL	R	T2 wecken
. . .	. . .			
unlock(b)			R	

Zweiphasen-Sperrprotokolle[1]

- **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

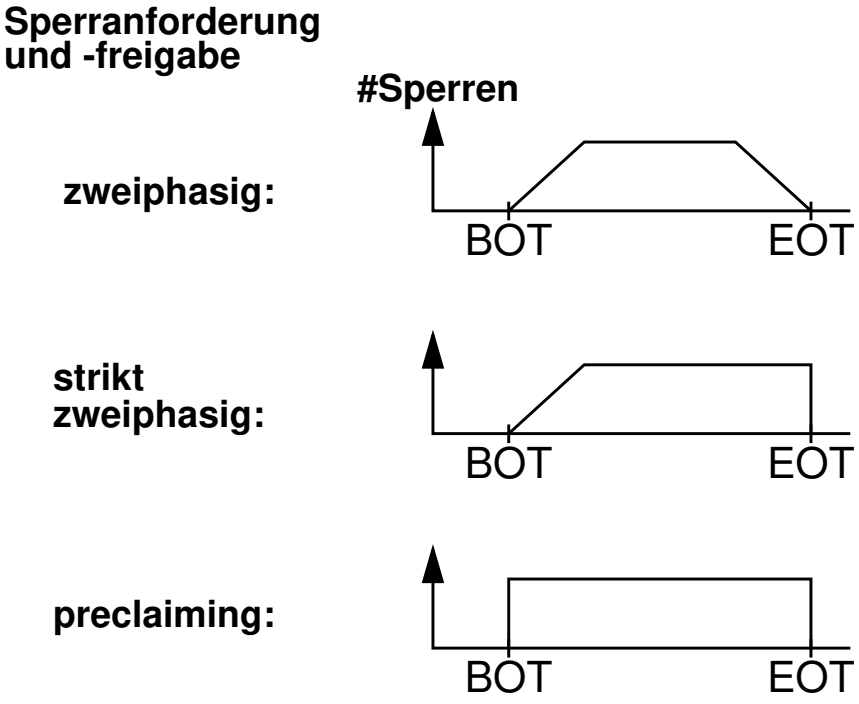
An der SQL-Schnittstelle ist die Sperranforderung und -freigabe nicht sichtbar!

---

1. Eswaran, K.P. et al.: The notions of consistency and predicate locks in a data base system, Comm. ACM 19:11, 1976, pp. 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
lock (b, X)		
read (b)		
unlock (a)		T2 wecken
	read (a)	
	write (a)	
	unlock (a)	
	commit	
unlock (b)		

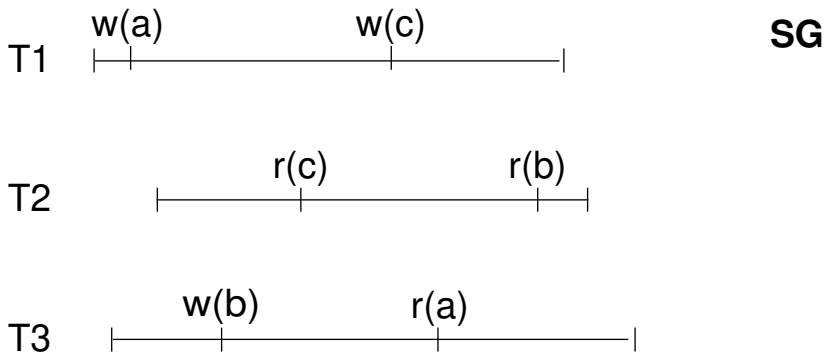
➥ Praktischer Einsatz erfordert **striktes 2PL-Protokoll**!

Verklemmungen (Deadlocks)

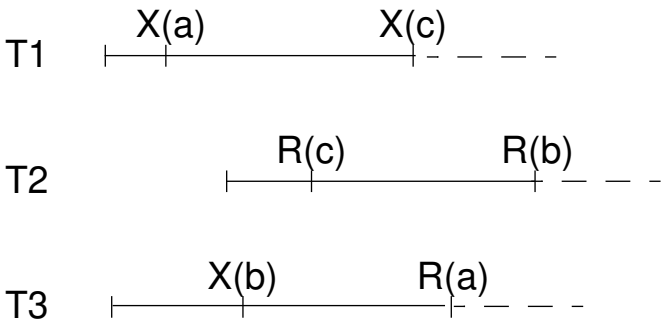
- **Striktes 2PL-Protokoll**
  - gibt alle Sperren erst bei Commit frei und
  - verhindert dadurch kaskadierendes Rücksetzen

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

- **Nicht-serialisierbare Historie**



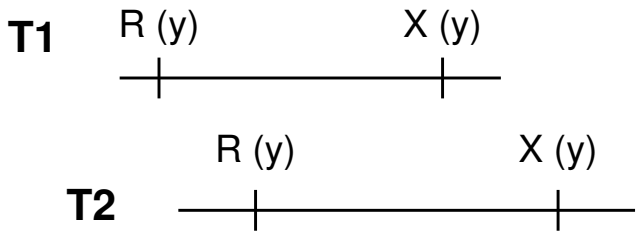
- **RX-Verfahren verhindert** das Auftreten einer nicht-serialisierbaren Historie, **aber nicht (immer) Deadlocks**



RUX-Sperrverfahren

- **Forderung**
  - Wahl des gemäß der Operation schwächst möglichen Sperrmodus
  - Möglichkeit der Sperrkonversion (upgrading), falls stärkerer Sperrmodus erforderlich
  - Anwendung: viele Objekte sind zu lesen, aber nur wenige zu aktualisieren

- **Problem: Sperrkonversionen**



- **Erweitertes Sperrverfahren:**
  - Ziel: Verhinderung von Konversions-Deadlocks
  - U-Sperre für Lesen mit Änderungsabsicht (Prüfmodus)
  - bei Änderung Konversion U $\rightarrow$ X, andernfalls U $\rightarrow$ R (downgrading)

- **Wirkungsweise**

RUX-Sperrverfahren (2)

- **Symmetrische Variante**
  - Was bewirkt eine Symmetrie bei U?

	R	U	X
R	+	+	-
U	+	-	-
X	-	-	-

- **Beispiel**

- **Unsymmetrie bei U**

	R	U	X
R	+	-	-
U	+	-	-
X	-	-	-

  - u. a. in DB2 eingesetzt

- **Beispiel**

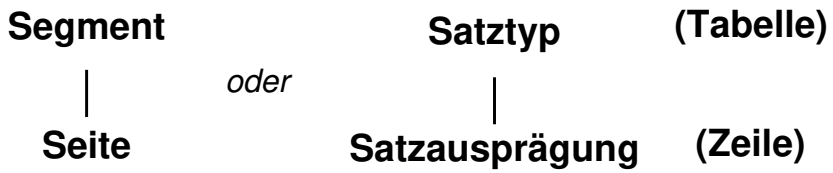
Hierarchische Sperrverfahren

- **Sperrgranulat bestimmt Parallelität/Aufwand:**

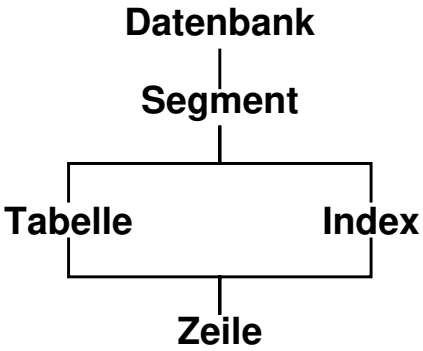
  Feines Granulat reduziert Sperrkonflikte, jedoch sind viele Sperren anzufordern und zu verwalten

- **Hierarchische Verfahren erlauben Flexibilität bei Wahl des Granulates ('multigranularity locking'), z. B. Synchronisation**
  - langer TA auf Tabellenebene
  - kurzer TA auf Zeilenebene

- Kommerzielle DBS unterstützen zumeist mindestens 2-stufige Objekthierarchie, z. B.

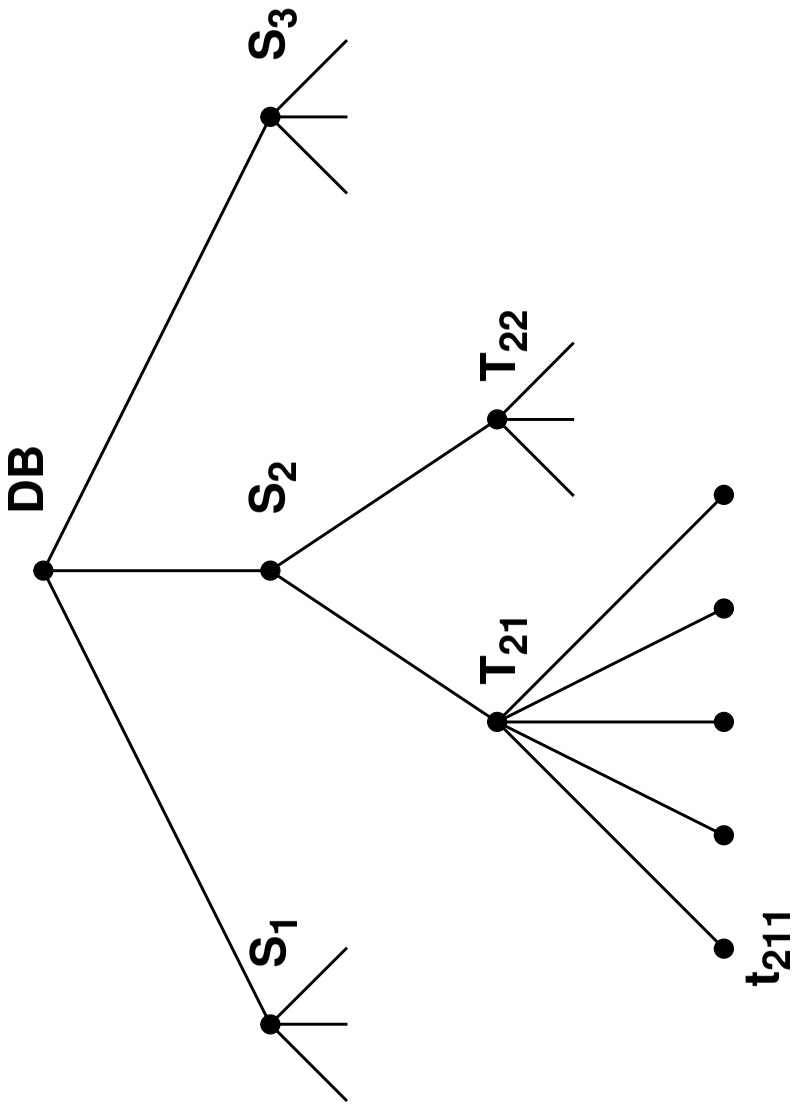


- Verfahren nicht auf reine Hierarchien beschränkt, sondern auch auf halbgeordnete Objektmengen erweiterbar.



- Verfahren erheblich komplexer als einfache Sperrverfahren (mehr Sperrmodi, Konversionen, Deadlock-Behandlung, ...)

Beispiel einer Sperrhierarchie



Hierarchische Sperrverfahren: Anwartschaftssperren

- **Mit R- und X-Sperre werden alle Nachfolgerknoten implizit mitgesperrt**

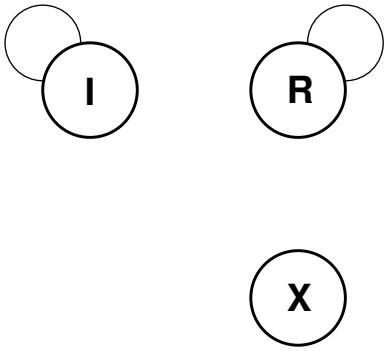
  ➥ Einsparungen möglich

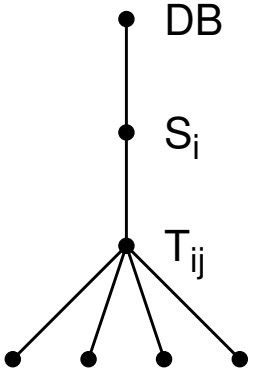
- **Alle Vorgängerknoten sind ebenfalls zu sperren, um Unverträglichkeiten zu vermeiden**

  ➥ Verwendung von Anwartschaftssperren ('intention locks')

- **Allgemeine Anwartschaftssperre (I-Sperre)**

	I	R	X
I	+	-	-
R	-	+	-
X	-	-	-





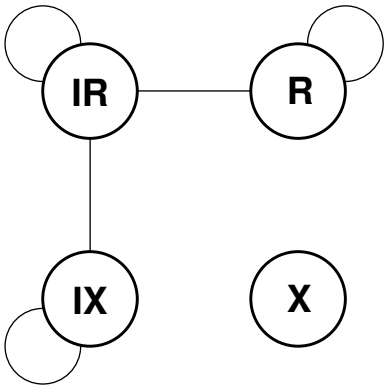
- Unverträglichkeit von I- und R-Sperren: zu restriktiv!

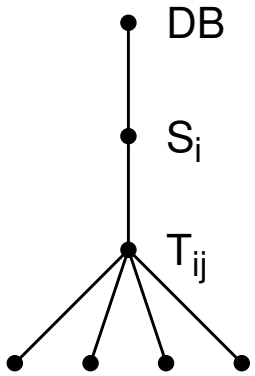
  ➥ zwei Arten von Anwartschaftssperren (IR und IX)

Anwartschaftssperren (2)

- **Anwartschaftssperren für Leser und Schreiber**

	IR	IX	R	X
IR	+	+	+	-
IX	+	+	-	-
R	+	-	+	-
X	-	-	-	-





- IR-Sperre (intent read), falls auf untergeordneten Objekten nur lesend zugegriffen wird, sonst IX-Sperre

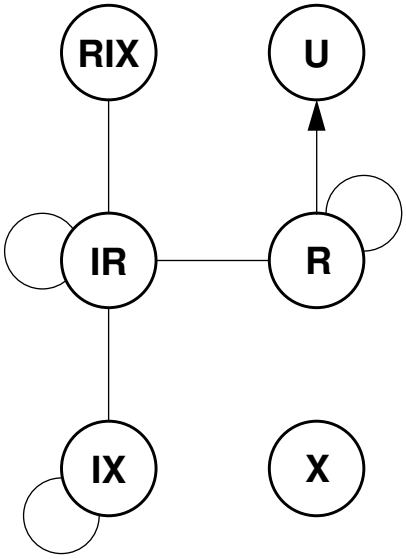
- Weitere Verfeinerung sinnvoll, um den Fall zu unterstützen, wo alle Sätze eines Satztyps gelesen und nur einige davon geändert werden sollen
  - X-Sperre auf Satztyp sehr restriktiv
  - IX-Sperre auf Satztyp verlangt Sperren jedes Satzes

  ➥ neuer Typ von Anwartschaftssperre: **RIX = R + IX**
  - sperrt das Objekt in R-Modus und verlangt
  - X-Sperren auf tieferer Hierarchieebene nur für zu ändernde Objekte

Anwartschaftssperren (3)

- **Vollständiges Protokoll der Anwartschaftssperren**
  - RIX gibt ein Leserecht auf den Knoten und seine Nachfolger. Weiterhin ist damit das Recht verbunden, auf Nachfolger-Knoten IX, U und X-Sperren anzufordern.
  - U gewährt ein Leserecht auf den Knoten und seine Nachfolger. Dieser Modus repräsentiert die Absicht, den Knoten in der Zukunft zu verändern. Bei Änderung Konversion U $\rightarrow$ X, sonst U $\rightarrow$ R.

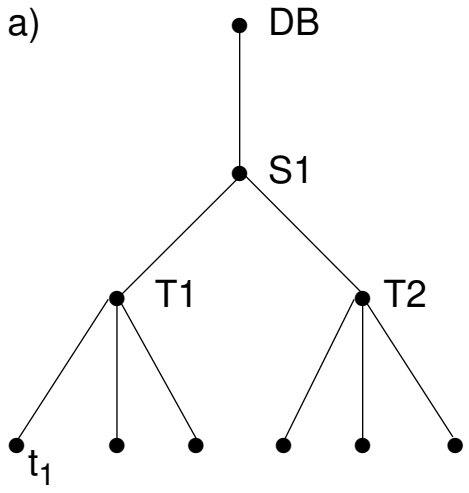
	IR	IX	R	RIX	U	X
IR	+	+	+	+	-	-
IX	+	+	-	-	-	-
R	+	-	+	-	-	-
RIX	+	-	-	-	-	-
U	-	-	+	-	-	-
X	-	-	-	-	-	-

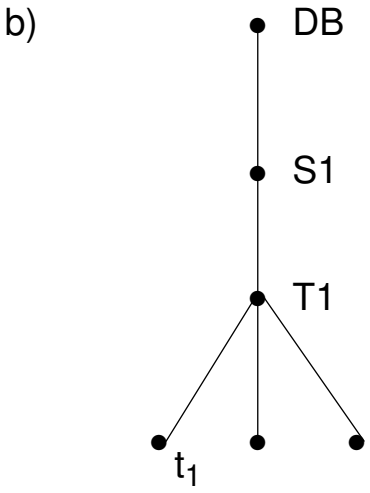


- **'Sperrdisziplin' erforderlich**
  - Sperranforderungen von der Wurzel zu den Blättern
  - Bevor T eine R- oder IR-Sperre für einen Knoten anfordert, muß sie für alle Vorgängerknoten IX- oder IR-Sperren besitzen
  - Bei einer X-, U-, RIX- oder IX-Anforderung müssen alle Vorgängerknoten in RIX oder IX gehalten werden
  - Sperrfreigaben von den Blättern zu der Wurzel
  - Bei EOT sind alle Sperren freizugeben

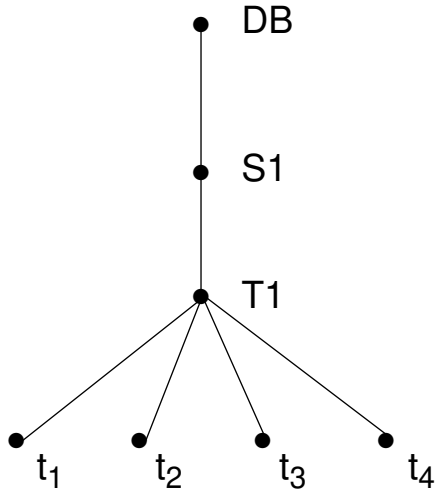
Hierarchische Sperrverfahren: Beispiele

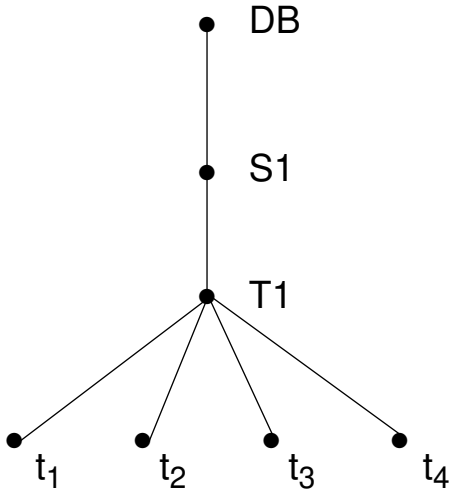
- **IR- und IX-Modus**
  - TA1 liest t_1 in T1
  - a) TA2 ändert Zeile in T2
  - b) TA3 liest T1





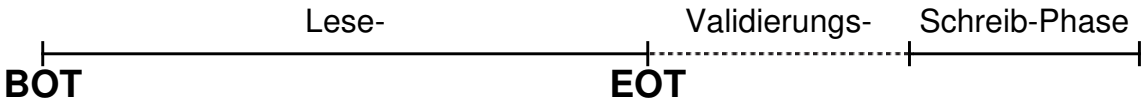
- **RIX-Modus**
  - TA1 liest alle Zeilen von T1 und ändert t_3
  - TA2 liest t_2 in T1



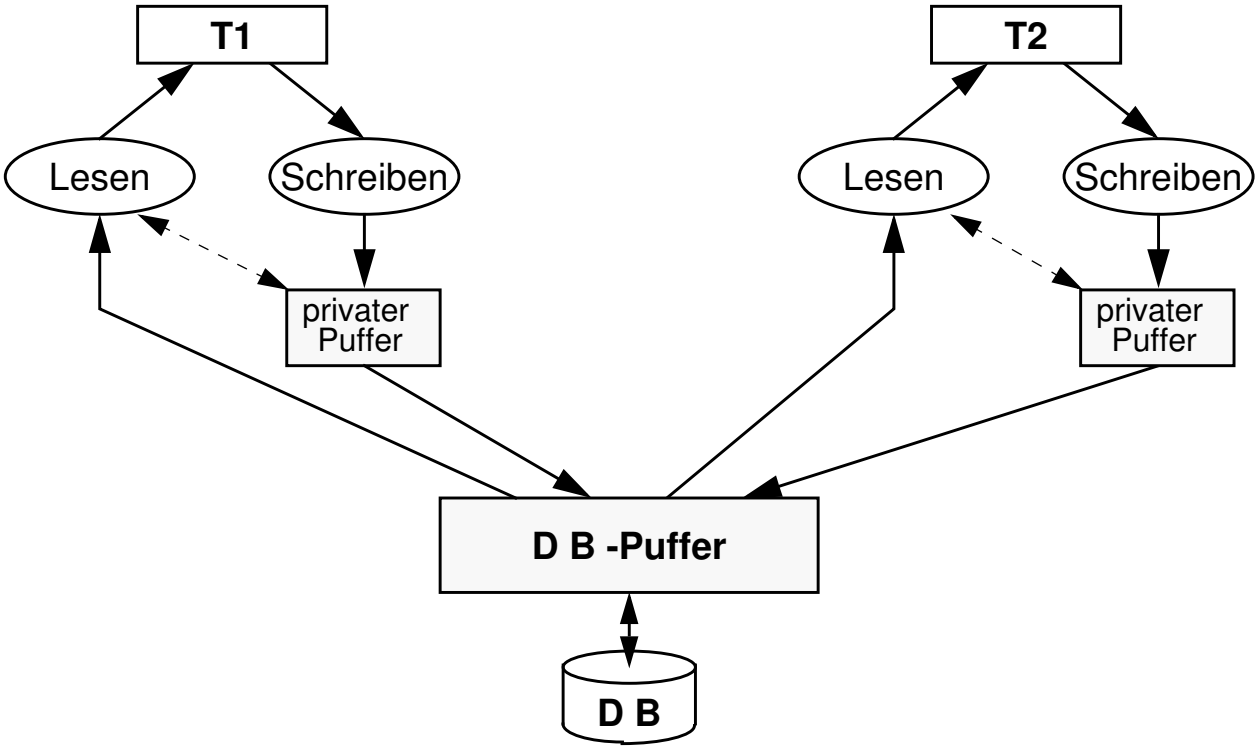


Optimistische Synchronisation (OCC)

- **3-phasige Verarbeitung:**



- **Lesephase**
  - eigentliche TA-Verarbeitung
  - Änderungen einer Transaktion werden in privatem Puffer durchgeführt



- **Validierungsphase**
  - Überprüfung, ob ein Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer der parallel ablaufenden Transaktionen passiert ist
  - Konfliktauflösung durch Zurücksetzen von Transaktionen
- **Schreibphase**
  - nur bei positiver Validierung
  - Lese-Transaktion ist ohne Zusatzaufwand beendet
  - Schreib-Transaktion schreibt hinreichende Log-Information und propagiert ihre Änderungen

Optimistische Synchronisation (2)

- **Grundannahme: geringe Konfliktwahrscheinlichkeit**

- **Allgemeine Eigenschaften von OCC:**
  + einfache TA-Rücksetzung
  + keine Deadlocks
  + potentiell höhere Parallelität als bei Sperrverfahren
  - mehr Rücksetzungen als bei Sperrverfahren
  - Gefahr des „Verhungerns” von TA

- **Durchführung der Validierungen:**
  Pro Transaktion werden geführt
  - Read-Set (RS) und
  - Write-Set (WS)

- **Forderung:**

  Eine TA kann nur erfolgreich validieren, wenn sie alle Änderungen von zuvor validierten TA gesehen hat

  ➥ Validierungsreihenfolge bestimmt Serialisierbarkeitsreihenfolge

- **Validierungsstrategien:**
  - **Backward Oriented (BOCC):**

    Validierung gegenüber bereits beendeten (Änderungs-) TA
  - **Forward Oriented (FOCC):**

    Validierung gegenüber laufenden TA

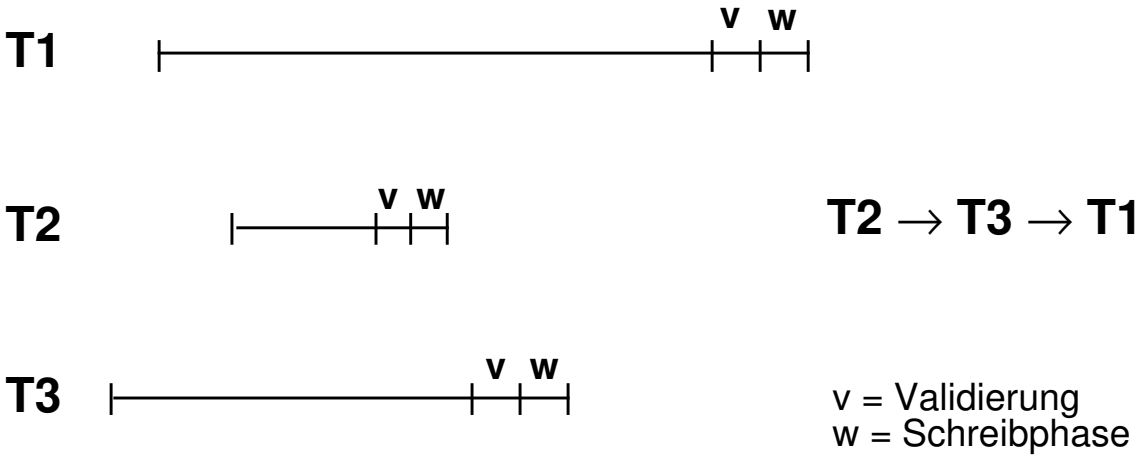
B O C C

- Erstes publizierte Verfahren zur optimistischen Synchronisation[1]

- **Validierung von Transaktion T:**

  BOCC-Test gegenüber allen Änderungs-TA T_j, die seit BOT von T erfolgreich validiert haben:

  IF $RS(T) \cap WS(T_j) \neq \emptyset$ THEN ABORT T
  ELSE SCHREIBPHASE



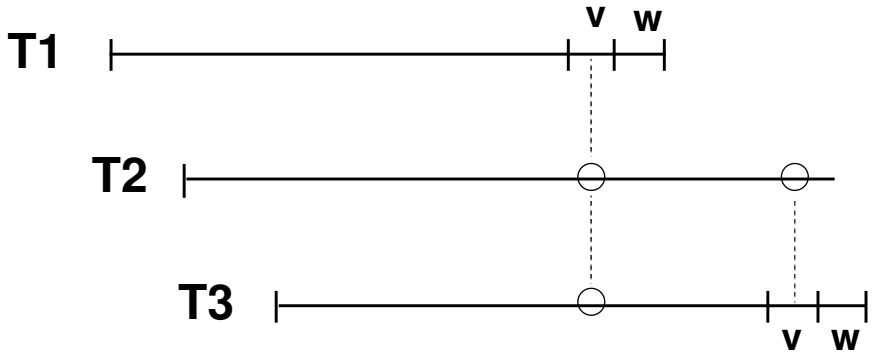
- **Nachteile/Probleme:**
  - unnötige Rücksetzungen wegen ungenauer Konfliktanalyse
  - Aufbewahren der Write-Sets beendeter TA erforderlich
  - hohe Anzahl von Vergleichen bei Validierung
  - Rücksetzung erst bei EOT ➥ viel unnötige Arbeit
  - Nur die validierende TA kann zurückgesetzt werden
    ➥ Gefahr von 'starvation'
  - hohes Rücksetzrisiko für lange TA und bei Hot-Spots

1. Kung, H.T., Robinson, J.T.: On optimistic method for concurrency control. ACM Trans. on Database Systems 6:2. 1981. 213-226

F O C C[1]

- Nur Änderungs-TA validieren gegenüber laufenden TA T_i

- **Validierungstest:** $WS(T) \cap RS(T_j) \stackrel{!}{=} \varnothing$



- **Vorteile:**
  - Wahlmöglichkeit des Opfers (Kill, Abort, Prioritäten, ...)
  - keine unnötigen Rücksetzungen
  - frühzeitige Rücksetzung möglich
    ➥ Einsparen unnötiger Arbeit
  - keine Aufbewahrung von Write-Sets,
    geringerer Validierungsaufwand als bei BOCC

- **Probleme:**
  - Während Validierungs- und Schreibphase muß WS (T) „gesperrt" sein, damit sich die RS (T_i) nicht ändern (keine Deadlocks damit möglich)
  - immer noch hohe Rücksetzrate möglich
  - Es kann immer nur einer TA Durchkommen definitiv zugesichert werden

1. Härder, T.: Observations on optimistic concurrency control schemes. Information Systems 9:2.1984. 111-120

Deadlock-Behandlung

- **Voraussetzungen für Deadlock:**
  - paralleler Zugriff
  - exklusive Zugriffsanforderungen
  - anfordernde TA besitzt bereits Objekte/Sperren
  - keine vorzeitige Freigabe von Objekten/Sperren *(non-preemption)*
  - zyklische Wartebeziehung zwischen zwei oder mehr TA

- **Lösungsmöglichkeiten:**

1. **Timeout-Verfahren**
   - TA wird nach festgelegter Wartezeit auf Sperre zurückgesetzt
   - problematische Bestimmung des Timeout-Wertes

2. **Deadlock-Verhütung *(Prevention)***
   - *keine Laufzeitunterstützung* zur Deadlock-Behandlung erforderlich
   - Beispiel: *Preclaiming* (in DBS i. allg. nicht praktikabel)

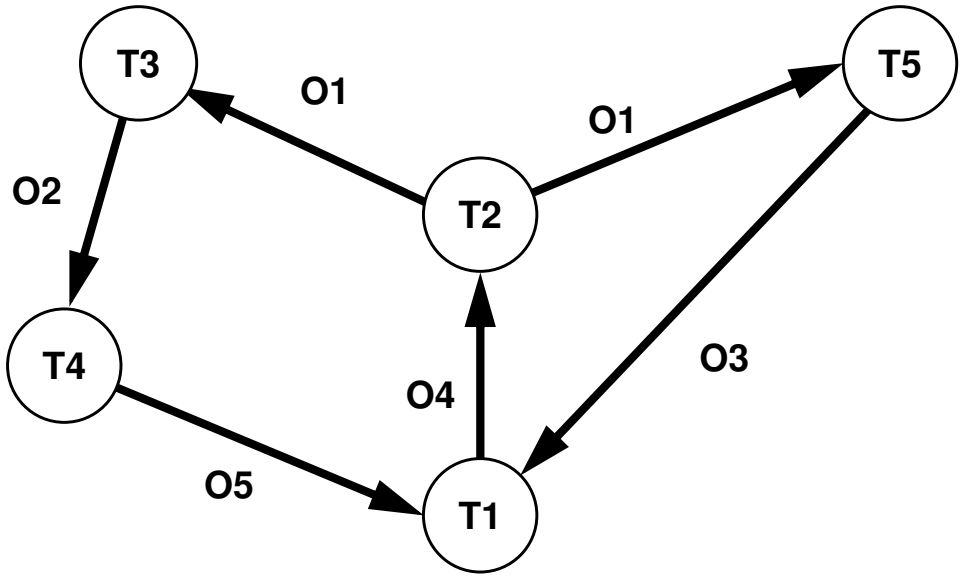
3. **Deadlock-Vermeidung *(Avoidance)***
   - Potentielle Deadlocks werden im voraus erkannt und durch entsprechende Maßnahmen vermieden

     $\Rightarrow$ *Laufzeitunterstützung nötig*

4. **Deadlock-Erkennung *(Detection)***

Deadlock-Erkennung

- Explizites Führen eines **Wartegraphen** *(wait-for graph)* und **Zyklensuche** zur Erkennung von Verklemmungen



- **Deadlock-Auflösung** durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter TA

  (z. B. Verursacher oder „billigste" TA zurücksetzen)

- **Zyklensuche** entweder
  - bei jedem Sperrkonflikt bzw.
  - verzögert (z. B. über Timeout gesteuert)

Konsistenzebenen

- **Serialisierbare Abläufe**
  - gewährleisten „automatisch“ Korrektheit des Mehrbenutzerbetriebs
  - erzwingen u. U. lange Blockierungszeiten paralleler Transaktionen
  - „Schwächere“ Konsistenzebene bei der Synchronisation von Leseoperationen erlaubt höhere Parallelitätsgrade und Reduktion von Blockierungen, erfordert aber Programmierdisziplin!

    ➥ **Inkaufnahme von Anomalien reduziert die TA-Behinderungen**

- **Konsistenzebenen** (basierend auf verkürzte Sperrdauern)

Ebene 3: Transaktion T sieht Konsistenzebene 3, wenn gilt:

a) T verändert keine schmutzigen Daten anderer Transaktionen

b) T gibt keine Änderungen vor EOT frei

c) T liest keine schmutzigen Daten anderer Transaktionen

d) Von T gelesene Daten werden durch andere Transaktionen erst nach EOT von T verändert

Ebene 2: Transaktion T sieht Konsistenzebene 2, wenn sie die Bedingungen a, b und c erfüllt

Ebene 1: Transaktion T sieht Konsistenzebene 1, wenn sie die Bedingungen a und b erfüllt

Ebene 0: Transaktion T sieht Konsistenzebene 0, wenn sie nur Bedingung a erfüllt

Konsistenzebenen (2)

- **RX-Sperrverfahren und Konsistenzebenen:**

Konsistenzebenen (3)

- ***Konsistenzebene 3*** (eigentlich KE 2,99)**:**
  - wünschenswert, jedoch oft viele Sperrkonflikte wegen langer Schreib- und Lesesperren

- ***Konsistenzebene 2:***
  - nur lange Schreibsperren, jedoch kurze Lesesperren
  - *'unrepeatable read'* möglich

- ***Konsistenzebene 1:***
  - lange Schreibsperren, keine Lesesperren
  - *'dirty read'* (und 'lost update') möglich

- ***Konsistenzebene 0:***
  - kurze Schreibsperren ('Chaos')

  ➥ Kommerzielle DBS empfehlen meist Konsistenzebene 2

- **Wahlangebot**

  Einige DBS (DB2, Tandem NonStop SQL, ...) bieten Wahlmöglichkeit zwischen:
  - 'repeatable read' (Ebene 3) und
  - 'cursor stability' (Ebene 2)

  Einige DBS bieten auch *BROWSE-Funktion*,
  d. h. Lesen ohne Setzen von Sperren (Ebene 1)

Konsistenzebenen (4)

- SQL erlaubt Wahl zwischen vier Konsistenzebenen (Isolation Level)

- **Konsistenzebenen sind durch die Anomalien bestimmt,**
  die jeweils in Kauf genommen werden:
  - Abgeschwächte Konsistenzanforderungen betreffen nur Leseoperationen!
  - **Lost Update** muß generell vermieden werden, d. h., W/W-Abhängigkeiten müssen stets beachtet werden

Konsistenz-ebene	**Anomalie**		
	Dirty Read	Non-Repeatable Read	Phantome
Read Uncommitted	**+**	**+**	**+**
Read Committed	**-**	**+**	**+**
Repeatable Read	**-**	**-**	**+**
Serializable	**-**	**-**	**-**

  - Default ist **Serialisierbarkeit** (serializable)

- **SQL-Anweisung zum Setzen der Konsistenzebene:**

```
SET TRANSACTION  [mode] [ISOLATION LEVEL level]
```

  - Transaktionsmodus: READ WRITE (Default) bzw. READ ONLY
  - **Beispiel:**
    SET TRANSACTION READ ONLY, ISOLATION LEVEL READ COMMITTED
  - READ UNCOMMITTED für Änderungstransaktionen unzulässig

Zusammenfassung

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

- **Korrektheitskriterium der Synchronisation: Serialisierbarkeit**

- **Theorie der Serialisierbarkeit**
  - einfaches Read/Write-Modell (Syntaktische Behandlung)
  - enorm gründlich erforscht
  - weitergehende Ansätze: Einbezug der Anwendungssemantik (Synchronisation von abstrakten Operationen auf Objekten)

- **Implementierung der Synchronisation: viele Verfahren**
  - Sperrverfahren sind universell einsetzbar
  - DBS-Standard: multiple Sperrgranulate durch hierarchische Sperrverfahren
  - Reine OCC-Verfahren erzeugen zuviele Rücksetzungen

- **Serialisierbare Abläufe**
  - gewährleisten „automatisch“ Korrektheit des Mehrbenutzerbetriebs
  - erzwingen u. U. lange Blockierungszeiten paralleler Transaktionen

- **Deadlock-Problem ist bei blockierenden Verfahren inhärent!**

- **Einführung von Konsistenzebenen**
  - zwei (geringfügig) unterschiedliche Ansätze
    - basierend auf Sperrdauer für R und X
    - basierend auf zu tolerierende “Fehler”
  - „Schwächere“ Synchronisation von Leseoperationen erlaubt höhere Parallelitätsgrade und Reduktion von Blockierungen, erfordert aber Programmierdisziplin!

    ➥ **Inkaufnahme von Anomalien reduziert die TA-Behinderungen**