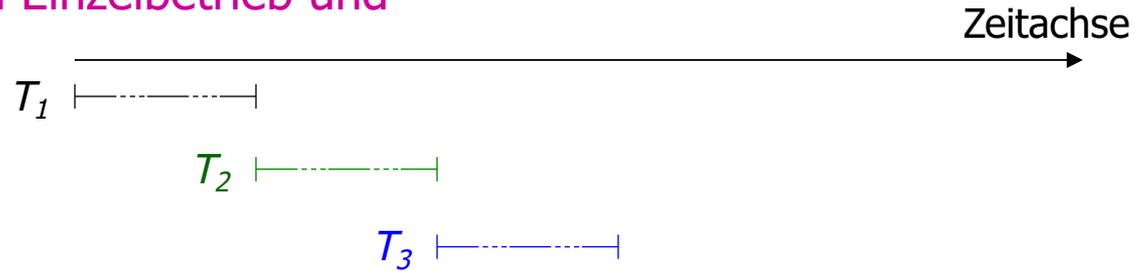
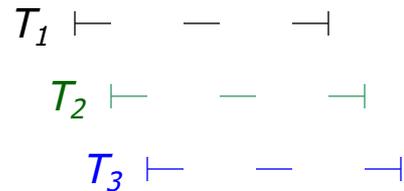


## Ausführung der drei Transaktionen $T_1$ , $T_2$ und $T_3$ :

(a) im Einzelbetrieb und



(b) im (verzahnten) Mehrbenutzerbetrieb (gestrichelte Linien repräsentieren Wartezeiten)



## Verlorengegangene Änderungen (*lost update*)

Schritt	$T_1$	$T_2$
1.	read( $A, a_1$ )	
2.	$a_1 := a_1 - 300$	
3.		read( $A, a_2$ )
4.		$a_2 := a_2 * 1.03$
5.		write( $A, a_2$ )
6.	write( $A, a_1$ )	
7.	read( $B, b_1$ )	
8.	$b_1 := b_1 + 300$	
9.	write( $B, b_1$ )	

## Abhängigkeit von nicht freigegebenen Änderungen

Schritt	$T_1$	$T_2$
1.	read(A,a <sub>1</sub> )	
2.	a <sub>1</sub> := a <sub>1</sub> - 300	
3.	write(A,a <sub>1</sub> )	
4.		read(A,a <sub>2</sub> )
5.		a <sub>2</sub> := a <sub>2</sub> * 1.03
6.		write(A,a <sub>2</sub> )
7.	read(B,b <sub>1</sub> )	
8.	...	
9.	abort	

# Fehler bei unkontrolliertem Mehrbenutzerbetrieb III

## Phantomproblem

 $T_1$  $T_2$ 

```
select sum(KontoStand)
```

```
from Konten
```

```
insert into Konten
```

```
values (C,1000,...)
```

```
select sum(Kontostand)
```

```
from Konten
```

Historie ist „äquivalent“ zu einer seriellen Historie  
dennoch parallele (verzahnte) Ausführung möglich

## Serialisierbare Historie von $T_1$ und $T_2$

Schritt	$T_1$	$T_2$
1.	<b>BOT</b>	
2.	read( $A$ )	
3.		<b>BOT</b>
4.		read( $C$ )
5.	write( $A$ )	
6.		write( $C$ )
7.	read( $B$ )	
8.	write( $B$ )	
9.	<b>commit</b>	
10.		read( $A$ )
11.		write( $A$ )
12.		<b>commit</b>

Serielle Ausführung von  $T_1$  vor  $T_2$ ,

also  $T_1 \mid T_2$

Schritt	$T_1$	$T_2$
1.	<b>BOT</b>	
2.	read( $A$ )	
3.	write( $A$ )	
4.	read( $B$ )	
5.	write( $B$ )	
6.	<b>commit</b>	
7.		<b>BOT</b>
8.		read( $C$ )
9.		write( $C$ )
10.		read( $A$ )
11.		write( $A$ )
12.		<b>commit</b>

# Nicht serialisierbare Historie

Schritt	$T_1$	$T_3$
1.	<b>BOT</b>	
2.	read( $A$ )	
3.	write( $A$ )	
4.		<b>BOT</b>
5.		read( $A$ )
6.		write( $A$ )
7.		read( $B$ )
8.		write( $B$ )
9.		<b>commit</b>
10.	read( $B$ )	
11.	write( $B$ )	
12.	<b>commit</b>	

# Zwei verzahnte Überweisungs-Transaktionen

Schritt	$T_1$	$T_3$
1.	<b>BOT</b>	
2.	read( $A, a_1$ )	
3.	$a_1 := a_1 - 50$	
4.	write( $A, a_1$ )	
5.		<b>BOT</b>
6.		read( $A, a_2$ )
7.		$a_2 := a_2 - 100$
8.		write( $A, a_2$ )
9.		read( $B, b_2$ )
10.		$b_2 := b_2 + 100$
11.		write( $B, b_2$ )
12.		<b>commit</b>
13.	read( $B, b_1$ )	
14.	$b_1 := b_1 + 50$	
15.	write( $B, b_1$ )	
16.	<b>commit</b>	

# Eine Überweisung ( $T_1$ ) und eine Zinsgutschrift ( $T_3$ )

Schritt	$T_1$	$T_3$
1.	<b>BOT</b>	
2.	read( $A, a_1$ )	
3.	$a_1 := a_1 - 50$	
4.	write( $A, a_1$ )	
5.		<b>BOT</b>
6.		read( $A, a_2$ )
7.		$a_2 := a_2 * 1.03$
8.		write( $A, a_2$ )
9.		read( $B, b_2$ )
10.		$b_2 := b_2 * 1.03$
11.		write( $B, b_2$ )
12.		<b>commit</b>
13.	read( $B, b_1$ )	
14.	$b_1 := b_1 + 50$	
15.	write( $B, b_1$ )	
16.	<b>commit</b>	

## „Formale“ Definition einer Transaktion

### Operationen einer Transaktion $T_i$

$r_i(A)$  zum Lesen des Datenobjekts  $A$ ,

$w_i(A)$  zum Schreiben des Datenobjekts  $A$ ,

$a_i$  zur Durchführung eines **aborts**,

$c_i$  zur Durchführung des **commit**.

**Konsistenzanforderung einer Transaktion  $T_i$**   
entweder **abort** oder **commit** aber nicht beides!

Falls  $T_i$  ein **abort** durchführt, müssen alle anderen Operationen  $p_j(A)$  vor  $a_i$  ausgeführt werden, also  $p_j(A) <_i a_i$ .

Analoges gilt für das **commit**, d.h.  $p_j(A) <_i c_i$  falls  $T_i$  „**committed**“.

Wenn  $T_i$  ein Datum  $A$  liest und auch schreibt, muss die Reihenfolge festgelegt werden, also entweder  $r_i(A) <_i w_i(A)$  oder  $w_i(A) <_i r_i(A)$ .

## Historie

$r_i(A)$  und  $r_j(A)$ : In diesem Fall ist die Reihenfolge der Ausführungen irrelevant, da beide TAs in jedem Fall denselben Zustand lesen. Diese beiden Operationen stehen also nicht in Konflikt zueinander, so dass in der Historie ihre Reihenfolge zueinander irrelevant ist.

$r_i(A)$  und  $w_j(A)$ : Hierbei handelt es sich um einen Konflikt, da  $T_j$  entweder den alten oder den neuen Wert von  $A$  liest. Es muss also entweder  $r_i(A)$  vor  $w_j(A)$  oder  $w_j(A)$  vor  $r_i(A)$  spezifiziert werden.

$w_i(A)$  und  $r_j(A)$ : analog

$w_i(A)$  und  $w_j(A)$ : Auch in diesem Fall ist die Reihenfolge der Ausführung entscheidend für den Zustand der Datenbasis; also handelt es sich um Konfliktoperationen, für die die Reihenfolge festzulegen ist.

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

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

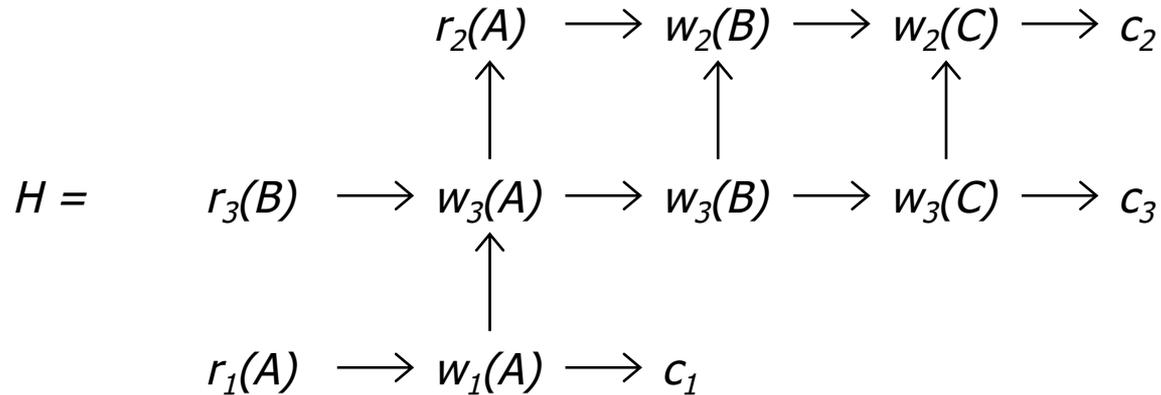
$$<_H \supseteq \bigcup_{i=1}^n <_i$$

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

-  $p <_H q$                       oder

-  $q <_H p$ .

## Beispiel-Historie für 3 TAs



# Äquivalenz zweier Historien

$H \equiv H'$  wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen

$r_1(A) \rightarrow r_2(C) \rightarrow w_1(A) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

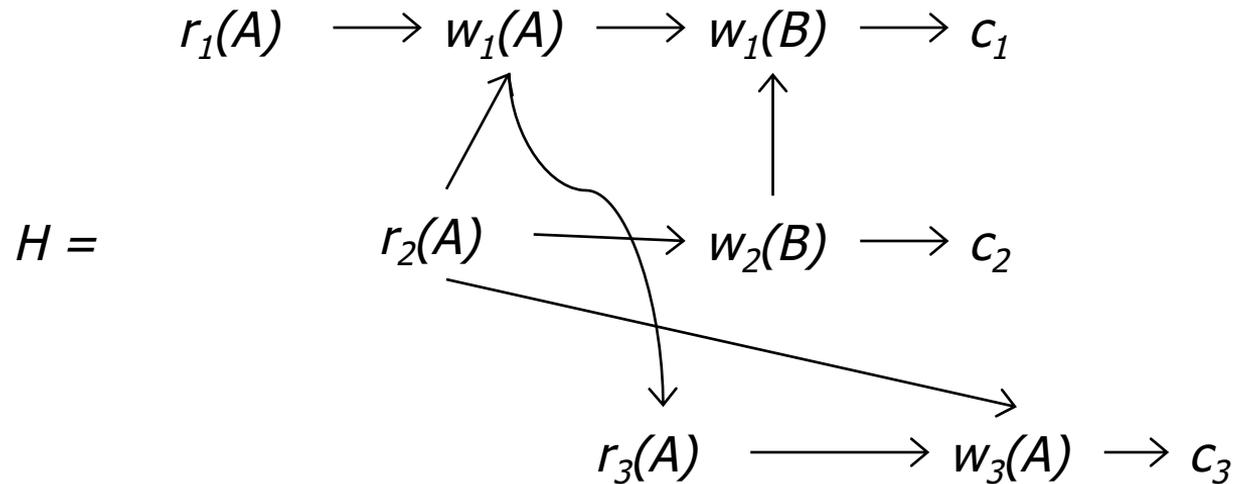
$r_1(A) \rightarrow w_1(A) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

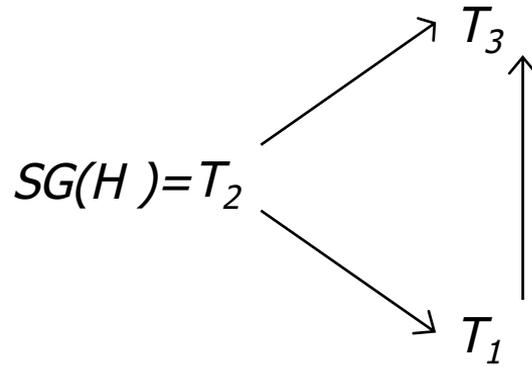
$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

Eine Historie ist *serialisierbar* wenn sie äquivalent zu einer seriellen Historie  $H_s$  ist.

## Historie und zugehöriger Serialisierbarkeitsgraph





- $w_1(A) \rightarrow r_3(A)$  der Historie  $H$  führt zur Kante  $T_1 \rightarrow T_3$  des SG
- weitere Kanten analog
- „Verdichtung“ der Historie

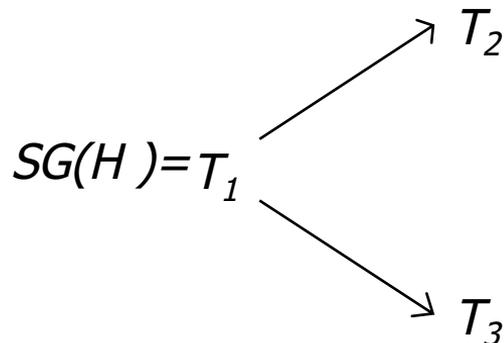
Eine Historie  $H$  ist genau dann *serialisierbar*, wenn der zugehörige Serialisierbarkeitsgraph  $SG(H)$  azyklisch ist.

## 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 Ordnung(en)

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

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

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

# Eigenschaften von Historien bezüglich der Recovery

## Terminologie

Wir sagen, dass in der Historie  $H$   $T_i$  von  $T_j$  liest, wenn folgendes gilt:

1.  $T_j$  schreibt mindestens ein Datum  $A$ , das  $T_i$  nachfolgend liest, also:
  - $w_j(A) <_H r_i(A)$
2.  $T_j$  wird (zumindest) nicht vor dem Lesevorgang von  $T_i$  zurückgesetzt, also:
  - $a_j \not<_H r_i(A)$
3. Alle anderen zwischenzeitlichen Schreibvorgänge auf  $A$  durch andere Transaktionen  $T_k$  werden vor dem Lesen durch  $T_i$  zurückgesetzt. Falls also ein  $w_k(A)$  mit  $w_j(A) < w_k(A) < r_i(A)$  existiert, so muss es auch ein  $a_k < r_i(A)$  geben.

# Eigenschaften von Historien bezüglich der Recovery

## Rücksetzbare Historien

Eine Historie heißt rücksetzbar, falls immer die schreibende Transaktion (in unserer Notation  $T_j$ ) vor der lesenden Transaktion ( $T_i$  genannt) ihr **commit** durchführt, also:

- $C_j <_H C_i$ .

Anders ausgedrückt: Eine Transaktion darf erst dann ihr **commit** durchführen, wenn alle Transaktionen, von denen sie gelesen hat, beendet sind.

# Eigenschaften von Historien bezüglich der Recovery

## Beispiel-Historie mit kaskadierendem Rücksetzen

Schritt	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
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.	$a_1(\text{abort})$				

### Historien ohne kaskadierendes Rücksetzen

Eine Historie vermeidet kaskadierendes Rücksetzen, wenn für je zwei TAs  $T_i$  und  $T_j$  gilt:

- $c_j <_H r_i(A)$  gilt, wann immer  $T_i$  ein Datum  $A$  von  $T_j$  liest.

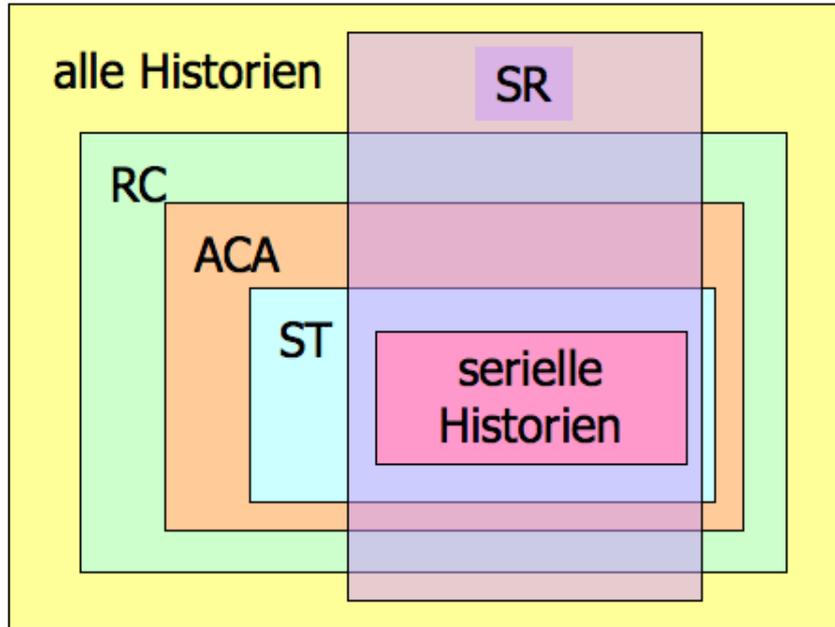
Eine Historie ist strikt, wenn für je zwei TAs  $T_i$  und  $T_j$  gilt: Wenn

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

Dann muss gelten:

- $a_j <_H o_i(A)$  oder
- $c_j <_H o_i(A)$

# Beziehungen zwischen den Klassen von Historien



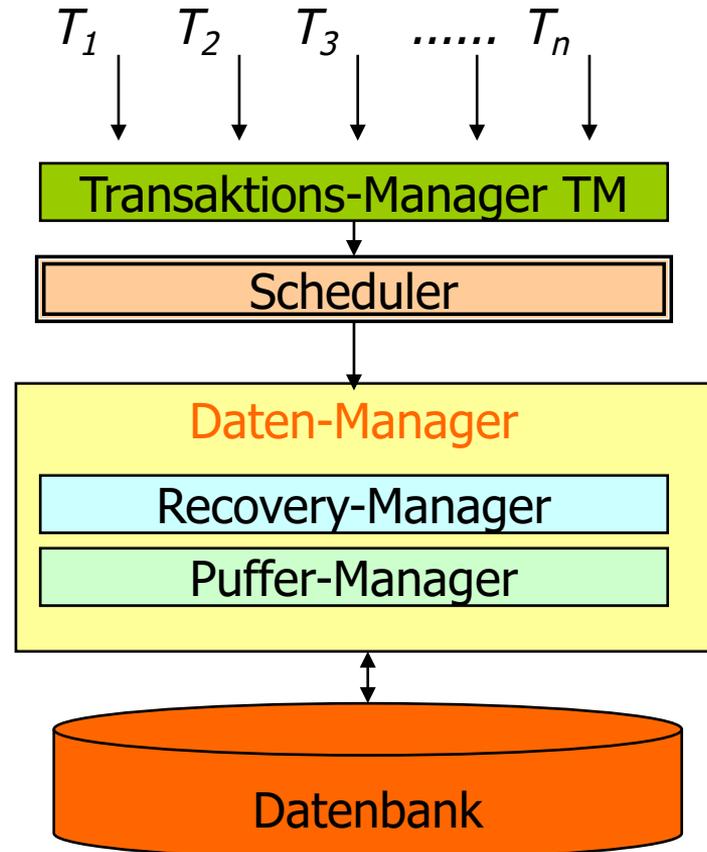
SR: serialisierbare Historien

RC: rücksetzbare Historien

ACA: *Historien ohne kaskadierendes Rücksetzen (avoiding cascading abort)*

ST: *strikte Historien*





# Sperrbasierte Synchronisation

## Zwei Sperrmodi

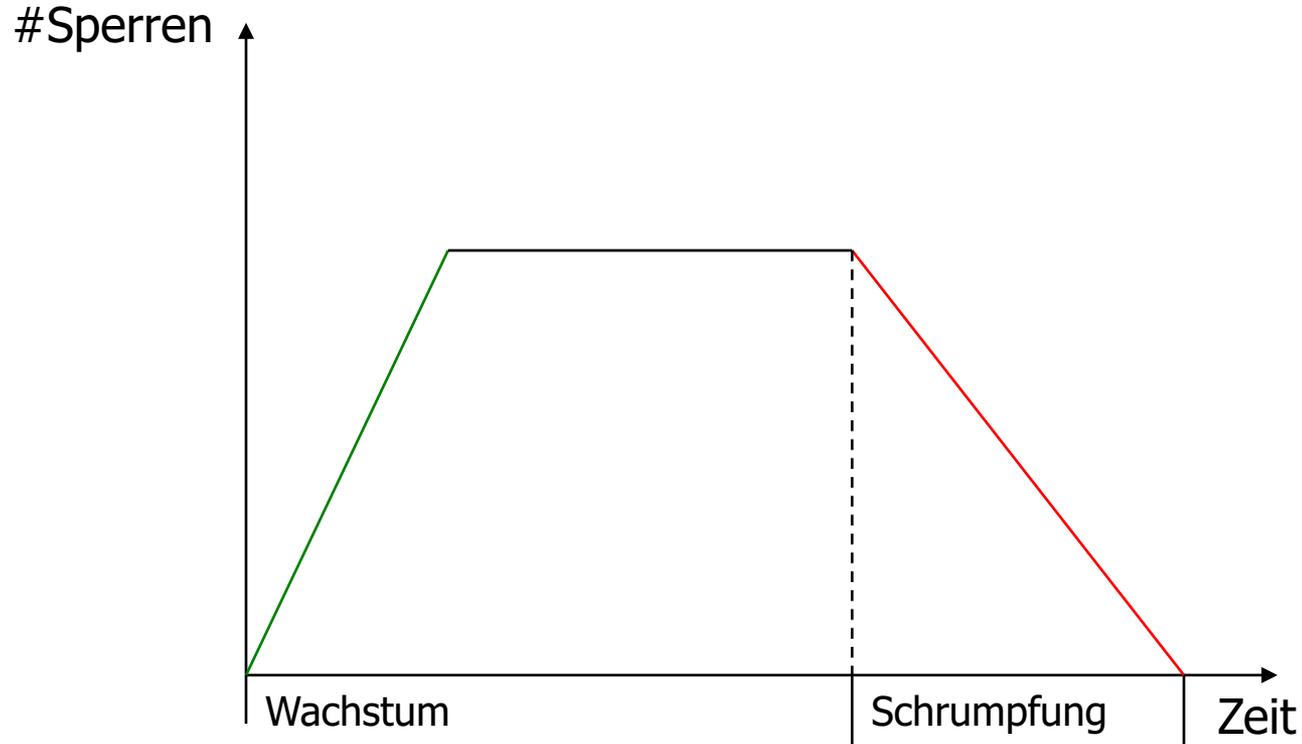
- S (shared, read lock, Lesesperre):
- X (exclusive, write lock, Schreibsperre):
- *Verträglichkeitsmatrix* (auch *Kompatibilitätsmatrix* genannt)

	NL	S	X
S	✓	✓	-
X	✓	-	-

# Zwei-Phasen-Sperrprotokoll: Definition

1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muss vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. eine Transaktion muss die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht – bis die Sperre gewährt werden kann.
4. Jede Transaktion durchläuft zwei Phasen:
  - Eine *Wachstumsphase*, in der sie Sperren anfordern, aber keine freigeben darf und
  - eine *Schrumpfphase*, in der sie ihre bisher erworbenen Sperren freigibt, aber keine weiteren anfordern darf.
5. Bei EOT (Transaktionsende) muss eine Transaktion alle ihre Sperren zurückgeben.

# Zwei-Phasen Sperrprotokoll: Grafik



## Verzahnung zweier TAs gemäß 2PL

$T_1$  modifiziert nacheinander die Datenobjekte  $A$  und  $B$  (z.B. eine Überweisung)

$T_2$  liest nacheinander dieselben Datenobjekte  $A$  und  $B$  (Z.B. zur Aufsummierung der beiden Kontostände).

# Verzahnung zweier TAs gemäß 2PL

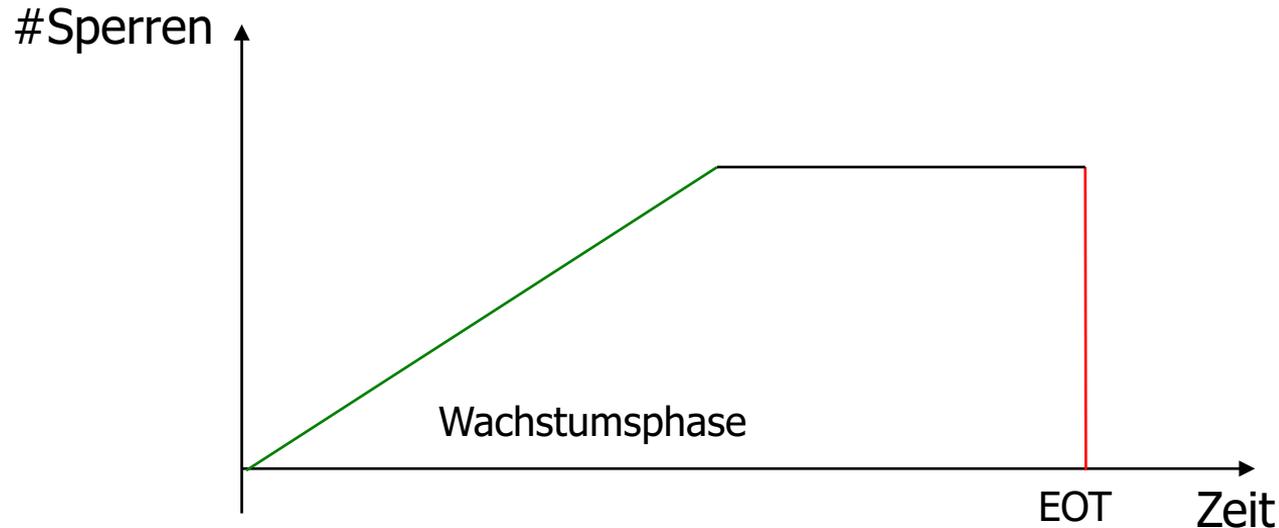
Schritt	$T_1$	$T_2$	Bemerkung
1.	<b>BOT</b>		
2.	<b>lockX(A)</b>		
3.	read(A)		
4.	write(A)		
5.		<b>BOT</b>	
6.		<b>lockS(A)</b>	$T_2$ muss warten
7.	<b>lockX(B)</b>		
8.	read(B)		
9.	<b>unlockX(A)</b>		$T_2$ wecken
10.		read(A)	
11.		<b>lockS(B)</b>	$T_2$ muss warten
12.	write(B)		
13.	<b>unlockX(B)</b>		$T_2$ wecken
14.		read(B)	
15.	<b>commit</b>		
16.		<b>unlockS(A)</b>	
17.		<b>unlockS(B)</b>	
18.		<b>commit</b>	

# Strenges Zwei-Phasen Sperrprotokoll

2PL schließt kaskadierendes Rücksetzen nicht aus

Erweiterung zum *strengen* 2PL:

- alle Sperren werden bis EOT gehalten
- damit ist kaskadierendes Rücksetzen ausgeschlossen



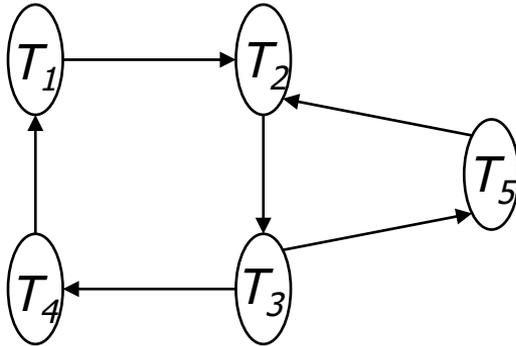
# Verklemmungen (Deadlocks)

## Ein verklemmter Schedule

Schritt	$T_1$	$T_2$	Bemerkung
1.	<b>BOT</b>		
2.	<b>lockX(A)</b>		
3.		<b>BOT</b>	
4.		<b>lockS(B)</b>	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	<b>lockX(B)</b>		$T_1$ muss warten auf $T_2$
9.		<b>lockS(A)</b>	$T_2$ muss warten auf $T_1$
10.	...	...	$\Rightarrow$ <i>Deadlock</i>

# Erkennungen von Verklemmungen

## Wartegraph mit zwei Zyklen

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$$
$$T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$$


- beide Zyklen können durch Rücksetzen von  $T_3$  „gelöst“ werden
- Zyklenerkennung durch Tiefensuche im Wartegraphen

# Verständnisfragen

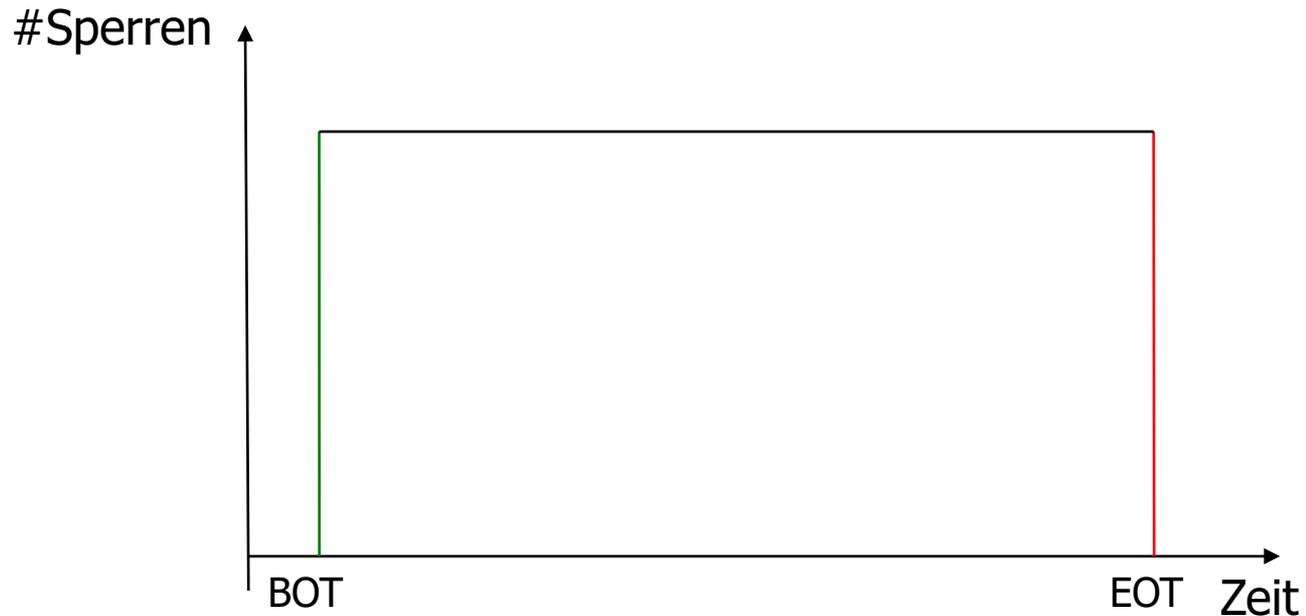
Commit-Reihenfolge

- Serielle Anordnung

Beziehung zwischen Wartegraph und SG

# Preclaiming zur Vermeidung von Verklemmungen

**Preclaiming in Verbindung mit dem strengen 2 PL-Protokoll**



# Verklemmungsvermeidung durch Zeitstempel

Jeder Transaktion wird ein eindeutiger Zeitstempel (TS) zugeordnet

ältere TAs haben einen kleineren Zeitstempel als jüngere TAs

TAs dürfen nicht mehr „bedingungslos“ auf eine Sperre warten

## wound-wait Strategie

$T_1$  will Sperre erwerben, die von  $T_2$  gehalten wird.

- Wenn  $T_1$  älter als  $T_2$  ist, wird  $T_2$  abgebrochen und zurückgesetzt, so dass  $T_1$  weiterlaufen kann.
- Sonst wartet  $T_1$  auf die Freigabe der Sperre durch  $T_2$ .

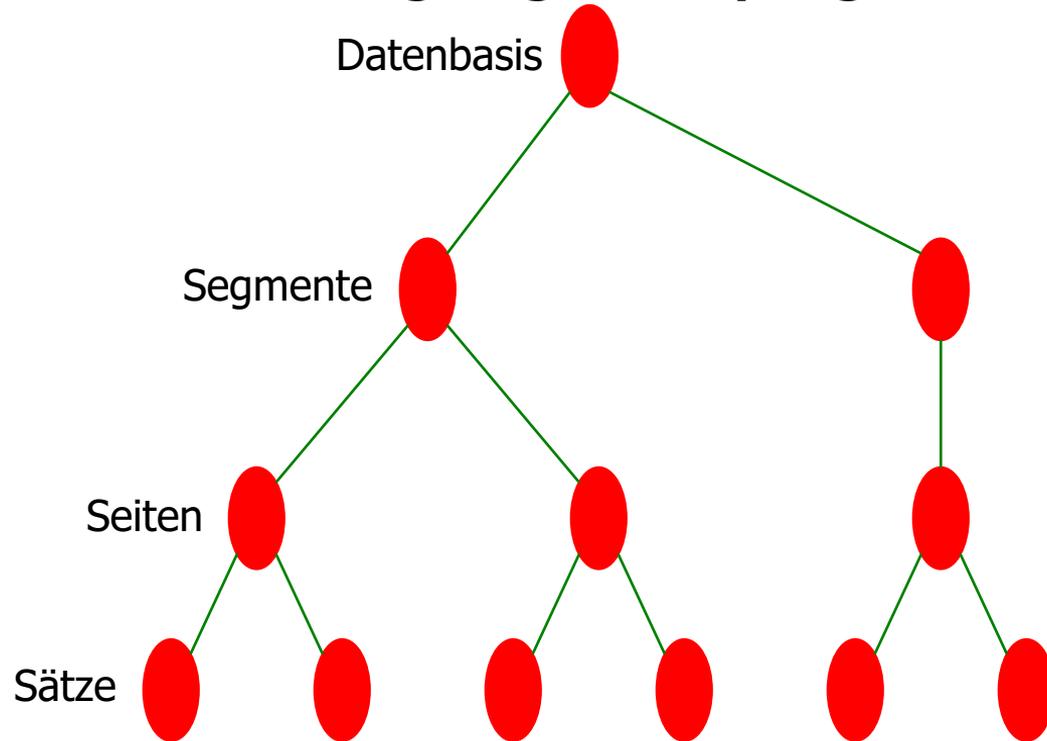
## wait-die Strategie

$T_1$  will Sperre erwerben, die von  $T_2$  gehalten wird.

- Wenn  $T_1$  älter als  $T_2$  ist, wartet  $T_1$  auf die Freigabe der Sperre.
- Sonst wird  $T_1$  abgebrochen und zurückgesetzt.



## Hierarchische Anordnung möglicher Sperrgranulate



# Erweiterte Sperrmodi

*NL*: keine Sperrung (no lock),

*S*: Sperrung durch Leser,

*X*: Sperrung durch Schreiber,

*IS* (intention share): Weiter unten in der Hierarchie ist eine Lesesperre (*S*) beabsichtigt,

*IX* (intention exclusive): Weiter unten in der Hierarchie ist eine Schreibsperre (*X*) beabsichtigt.

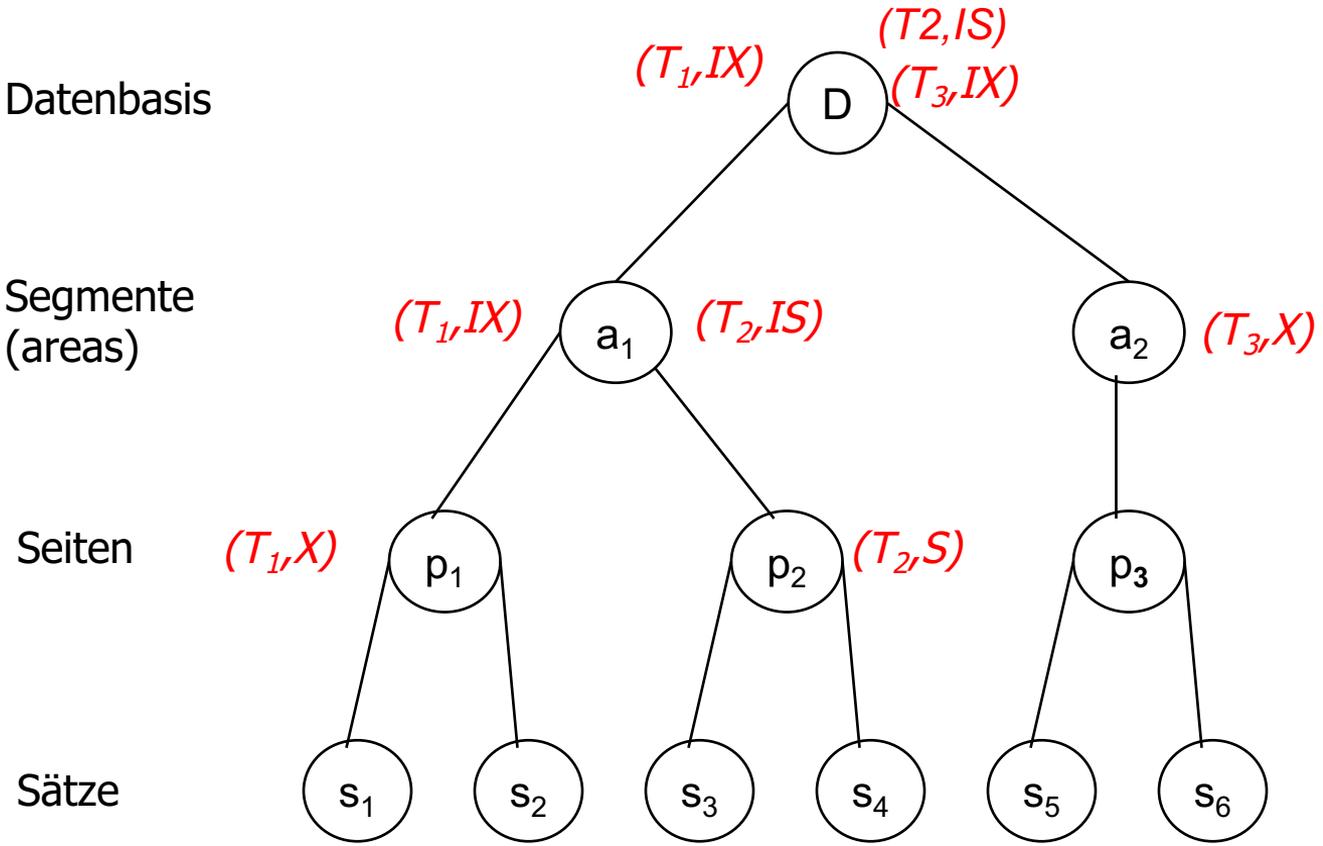
## Kompatibilitätsmatrix

	<i>NL</i>	<i>S</i>	<i>X</i>	<i>IS</i>	<i>IX</i>
<i>S</i>	✓	✓	-	✓	-
<i>X</i>	✓	-	-	-	-
<i>IS</i>	✓	✓	-	✓	✓
<i>IX</i>	✓	-	-	✓	✓

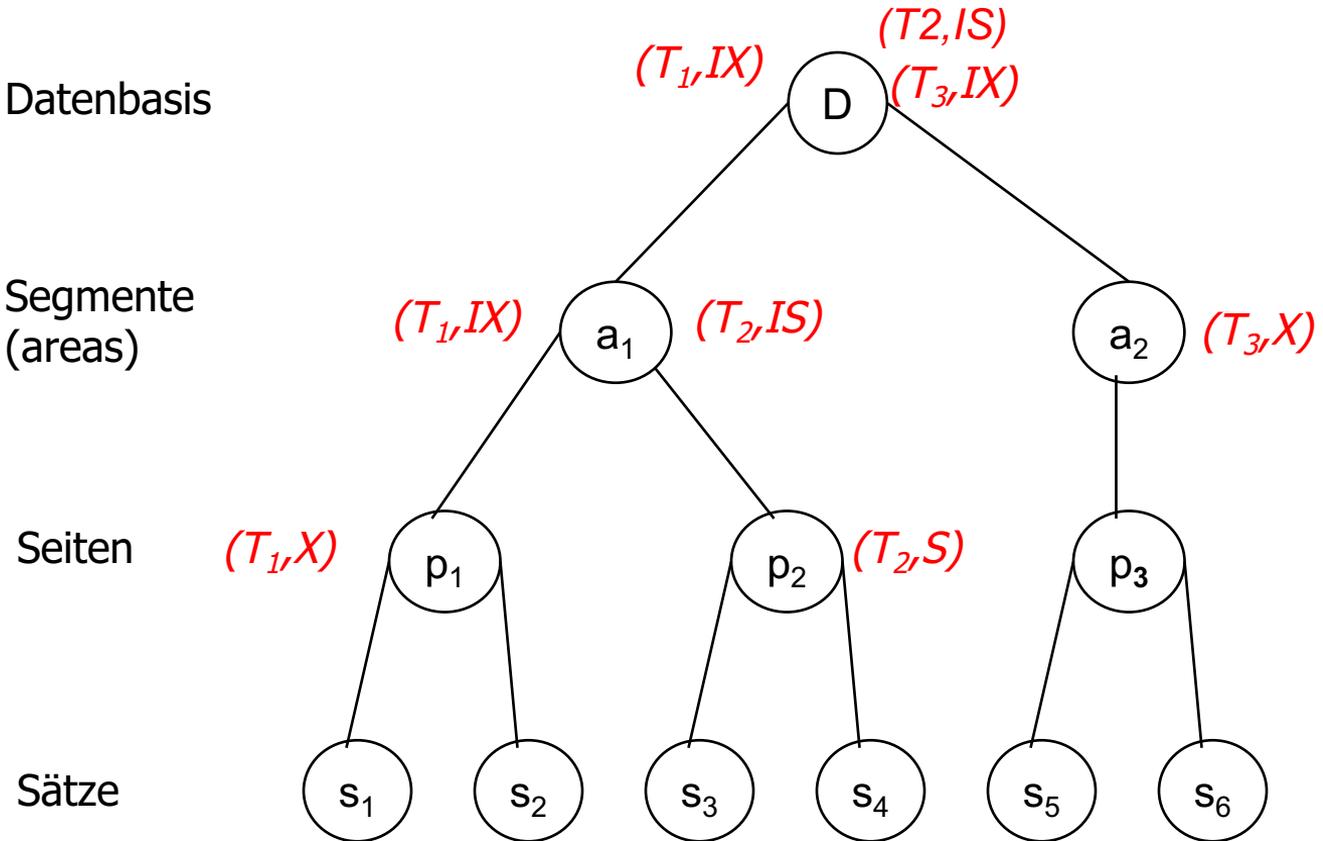
## Sperrprotokoll des MGL

1. Bevor ein Knoten mit *S* oder *IS* gesperrt wird, müssen alle Vorgänger in der Hierarchie vom Sperrer (also der Transaktion, die die Sperre anfordert) im *IX*- oder *IS*-Modus gehalten werden.
2. Bevor ein Knoten mit *X* oder *IX* gesperrt wird, müssen alle Vorgänger vom Sperrer im *IX*-Modus gehalten werden.
3. Die Sperren werden von unten nach oben (bottom up) freigegeben, so dass bei keinem Knoten die Sperre freigegeben wird, wenn die betreffende Transaktion noch Nachfolger dieses Knotens gesperrt hat.

# Datenbasis-Hierarchie mit Sperren

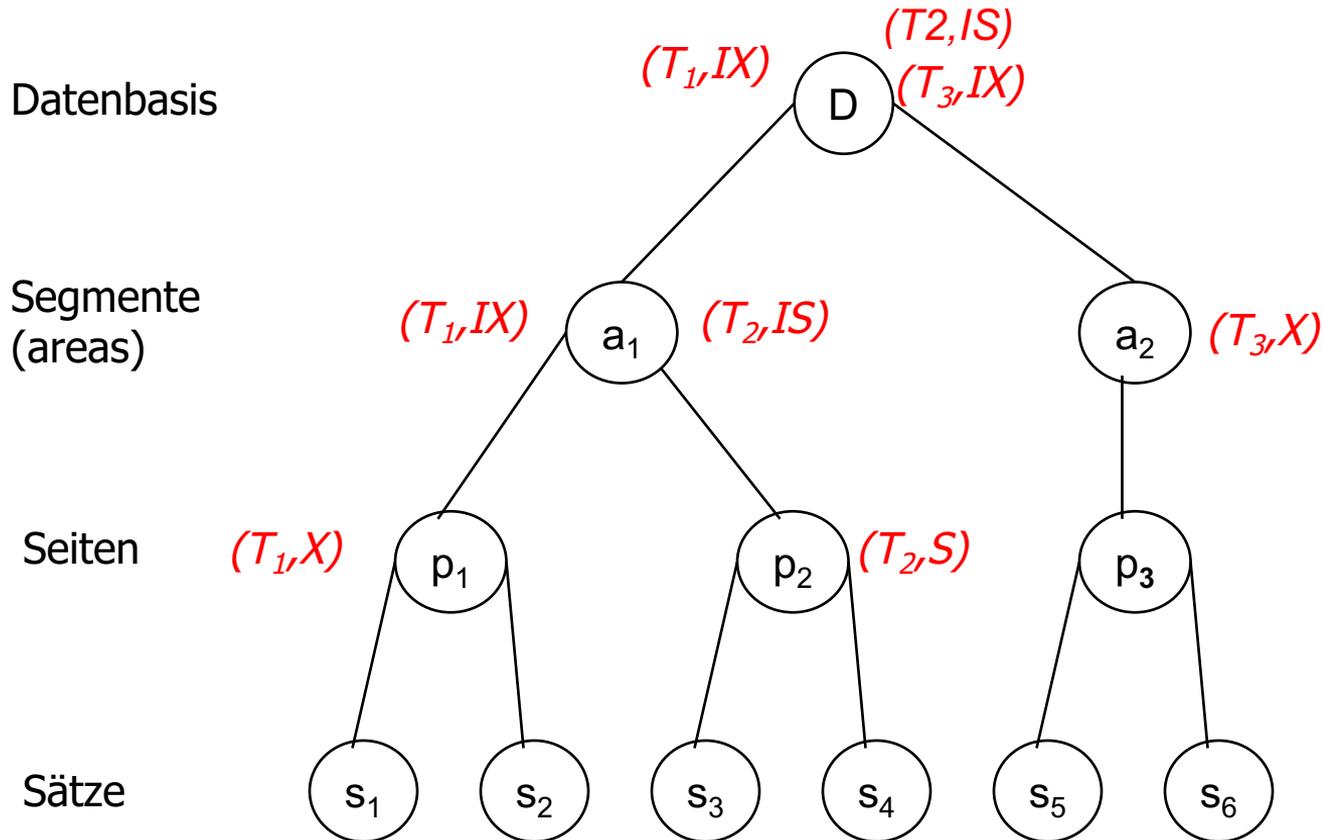


# Datenbasis-Hierarchie mit Sperren



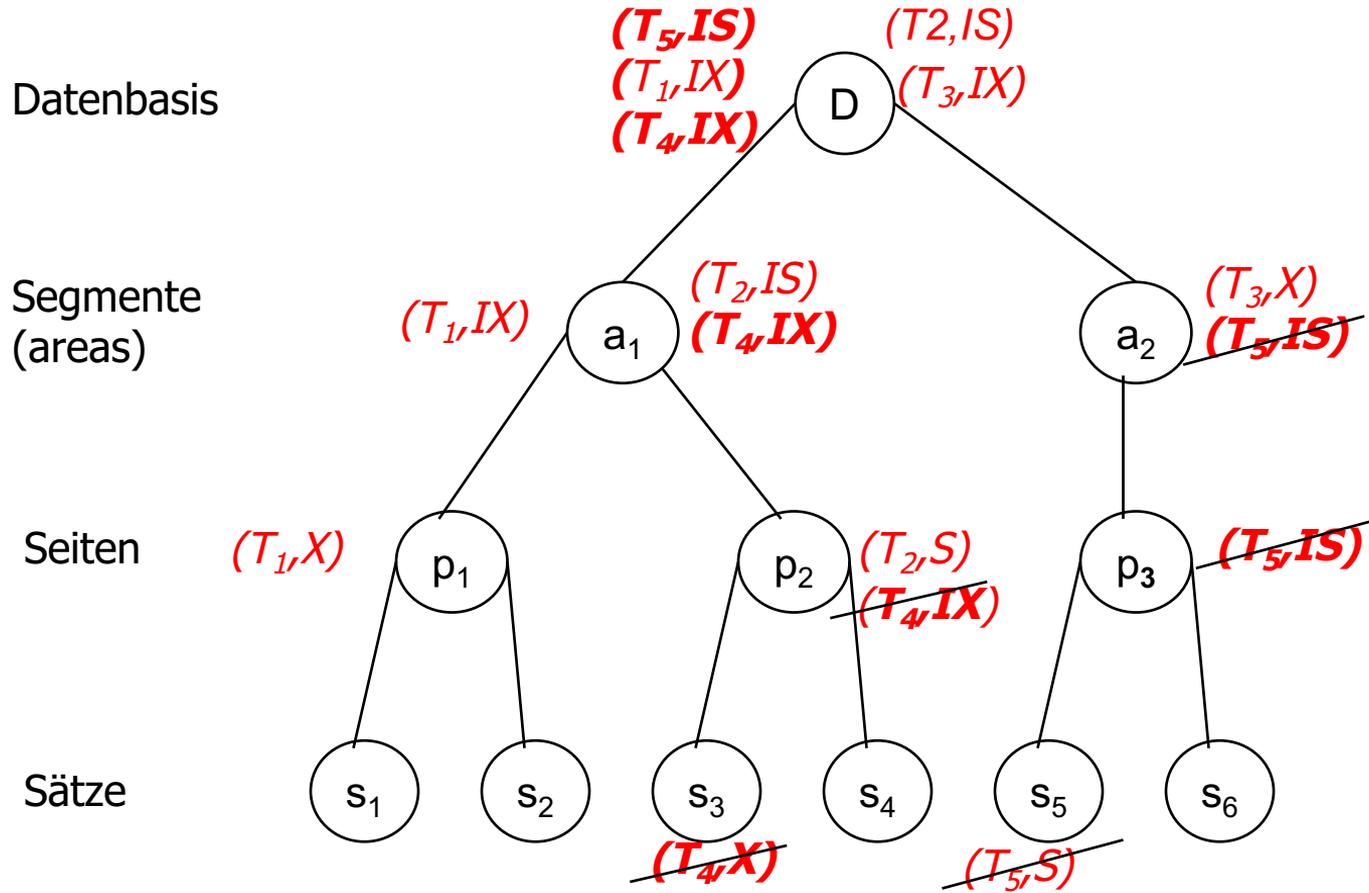
# Datenbasis-Hierarchie mit Sperren

( $T_4$  will  $s_3$  ändern,  $T_5$  will  $s_5$  lesen, was passiert?)



# Datenbasis-Hierarchie mit **blockierten** Transaktionen

( $T_4$  will  $s_3$  ändern,  $T_5$  will  $s_5$  lesen, was passiert?)



# Datenbasis-Hierarchie

## mit blockierten Transaktionen

- die TAs  $T_4$  und  $T_5$  sind blockiert (warten auf Freigabe von Sperren)
- es gibt aber in diesem Beispiel (noch) keine Verklemmung
- Verklemmungen sind aber auch bei MGL möglich

# Einfüge- und Löschoperationen, Phantome

Vor dem Löschen eines Objekts muss die Transaktion eine **X-Sperre** für dieses Objekt erwerben. Man beachte aber, dass eine andere TA, die für dieses Objekt ebenfalls eine Sperre erwerben will, diese nicht mehr erhalten kann, falls die Löschtransaktion erfolgreich (mit **commit**) abschließt. Beim Einfügen eines neuen Objekts erwirbt die einfügende Transaktion eine **X-Sperre**.

# Phantomprobleme

$T_1$	$T_2$
<pre><b>select count(*)</b> <b>from</b> prüfen <b>where</b> Note <b>between</b> 1 <b>and</b> 2;</pre>	
<pre><b>select count(*)</b> <b>from</b> prüfen <b>where</b> Note <b>between</b> 1 <b>and</b> 2</pre>	<pre><b>insert into</b> prüfen     <b>values</b>(19555, 5001, 2137, 1);</pre>

# Phantomprobleme

Das Problem lässt sich dadurch lösen, dass man zusätzlich zu den Tupeln auch den Zugriffsweg, auf dem man zu den Objekten gelangt ist, sperrt.

Wenn also ein Index für das Attribut *Note* existiert, würde der Indexbereich [1,2] für  $T_1$  mit einer *S-Sperre* belegt.

Wenn jetzt also Transaktion  $T_2$  versucht, das Tupel [29555, 5001, 2137, 1] in *prüfen* einzufügen, wird die TA blockiert



# Zeitstempel-basierende Synchronisation

Jedem Datum  $A$  in der Datenbasis werden bei diesem Synchronisationsverfahren zwei Marken zugeordnet:

1.  $readTS(A)$ :
2.  $writeTS(A)$ :

## Synchronisationsverfahren

- $T_i$  will  $A$  lesen, also  $r_i(A)$ 
  - Falls  $TS(T_i) < writeTS(A)$  gilt, haben wir ein Problem:
    - ★ Die Transaktion  $T_i$  ist älter als eine andere Transaktion, die  $A$  schon geschrieben hat.
    - ★ Also muss  $T_i$  zurückgesetzt werden.
  - Anderenfalls, wenn also  $TS(T_i) \geq writeTS(A)$  gilt, kann  $T_i$  ihre Leseoperation durchführen und die Marke  $readTS(A)$  wird auf  $max(TS(T_i), readTS(A))$  gesetzt.

# Zeitstempel-basierende Synchronisation

## Synchronisationsverfahren

- $T_i$  will  $A$  schreiben, also  $w_i(A)$ 
  - Falls  $TS(T_i) < readTS(A)$  gilt, gab es eine jüngere Lesetransaktion, die den neuen Wert von  $A$ , den  $T_i$  gerade beabsichtigt zu schreiben, hätte lesen müssen. Also muss  $T_i$  zurückgesetzt werden.
  - Falls  $TS(T_i) < writeTS(A)$  gilt, gab es eine jüngere Schreibtransaktion. D.h.  $T_i$  beabsichtigt einen Wert einer jüngeren Transaktion zu überschreiben. Das muss natürlich verhindert werden, so dass  $T_i$  auch in diesem Fall zurückgesetzt werden muss.
  - Anderenfalls darf  $T_i$  das Datum  $A$  schreiben und die Marke  $writeTS(A)$  wird auf  $TS(T_i)$  gesetzt.

## 1. *Lese*phase:

- In dieser Phase werden alle Operationen der Transaktion ausgeführt – also auch die Änderungsoperationen.
- Gegenüber der Datenbasis tritt die Transaktion in dieser Phase aber nur als Leser in Erscheinung, da alle gelesenen Daten in lokalen Variablen der Transaktion gespeichert werden.
- alle Schreiboperationen werden zunächst auf diesen lokalen Variablen aufgeführt.

## 2. *Validierungs*phase:

- In dieser Phase wird entschieden, ob die Transaktion möglicherweise in Konflikt mit anderen Transaktionen geraten ist.
- Dies wird anhand von Zeitstempeln entschieden, die den Transaktionen in der Reihenfolge zugewiesen werden, in der sie in die Validierungsphase eintreten.

## 3. *Schreib*phase:

- Die Änderungen der Transaktionen, bei denen die Validierung positiv verlaufen ist, werden in dieser Phase in die Datenbank eingebracht.



# Validierung bei der optimistischen Synchronisation

**Vereinfachende Annahme:** Es ist immer nur eine TA in der Validierungsphase!

Wir wollen eine Transaktion  $T_j$  validieren. Die Validierung ist erfolgreich falls für **alle** älteren Transaktionen  $T_a$  – also solche die früher ihre Validierung abgeschlossen haben – eine der beiden folgenden Bedingungen gelten:

1.  $T_a$  war zum Beginn der Transaktion  $T_j$  schon abgeschlossen – einschließlich der Schreibphase.
2. Die Menge der von  $T_a$  geschriebenen Datenelemente, genannt  $WriteSet(T_a)$  enthält keine Elemente der Menge der gelesenen Datenelemente von  $T_j$ , genannt  $ReadSet(T_j)$ . *Es muss also gelten:*

$$WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$$

# Validierung bei der Snapshot Isolation (SI)

**Vorsicht: SI garantiert nicht die Serialisierbarkeit – wird aber heute oft für hoch-skalierende Systeme verwendet**

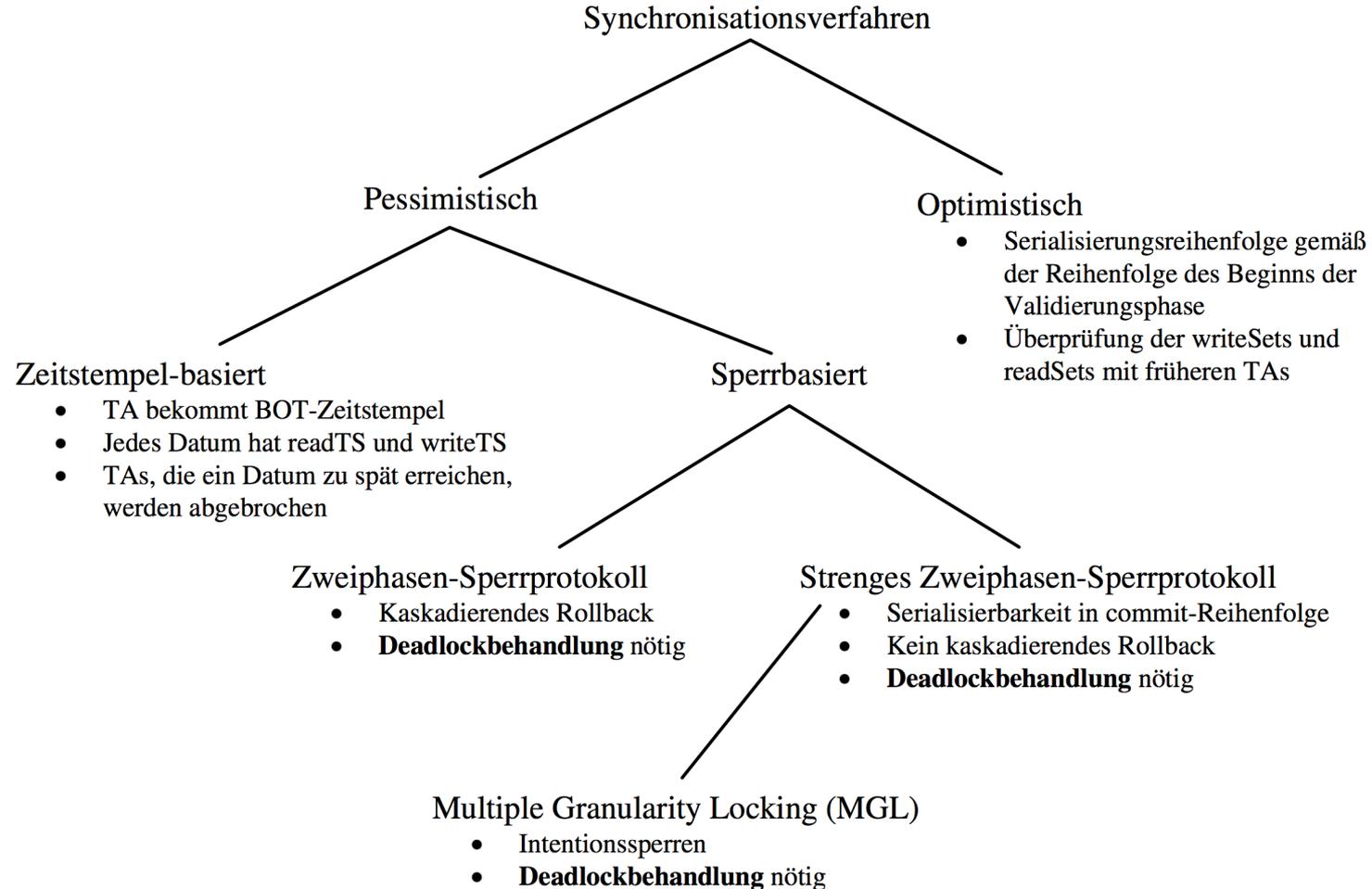
Wir wollen eine Transaktion  $T_j$  validieren. Die Validierung ist erfolgreich falls für **alle** älteren Transaktionen  $T_a$  – also solche die früher ihre Validierung abgeschlossen haben – eine der beiden folgenden Bedingungen gelten:

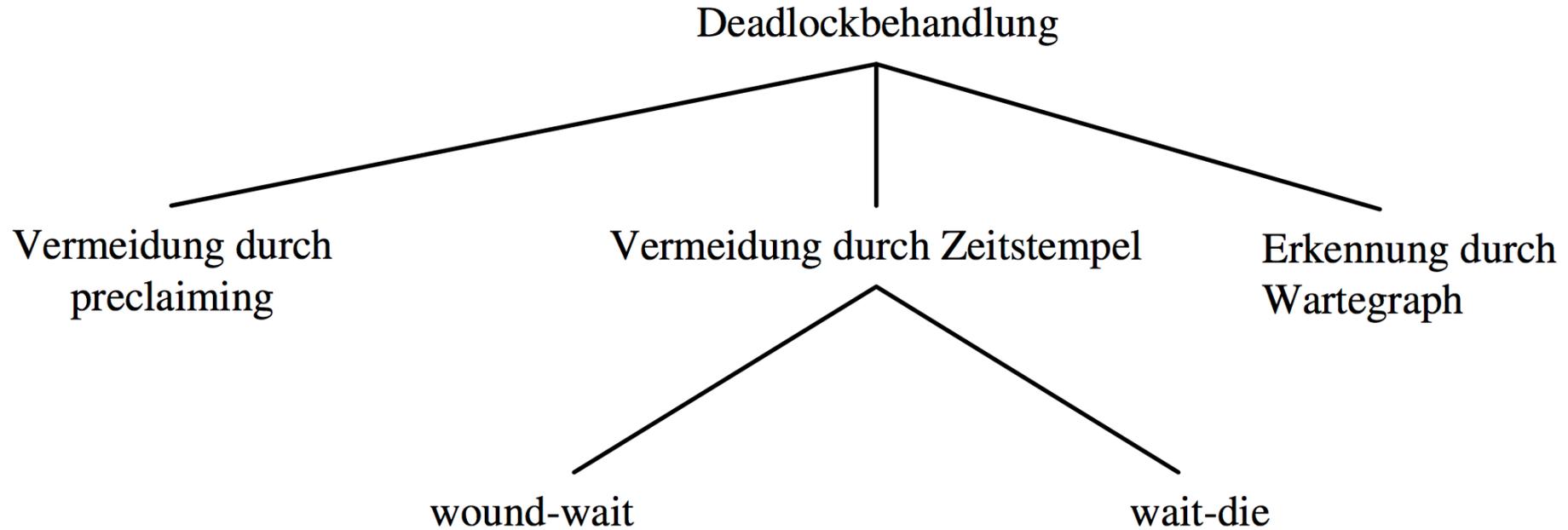
1.  $T_a$  war zum Beginn der Transaktion  $T_j$  schon abgeschlossen – einschließlich der Schreibphase.
2. Die Menge der von  $T_a$  geschriebenen Datenelemente, genannt  $WriteSet(T_a)$  enthält keine Elemente der Menge der gelesenen geschriebenen Datenelemente von  $T_j$ , genannt  $WriteSet(T_j)$ . Es muss also gelten:

$$ReadSet(T_j) \cap WriteSet(T_a) = \emptyset$$



# Klassifikation der Synchronisationsverfahren



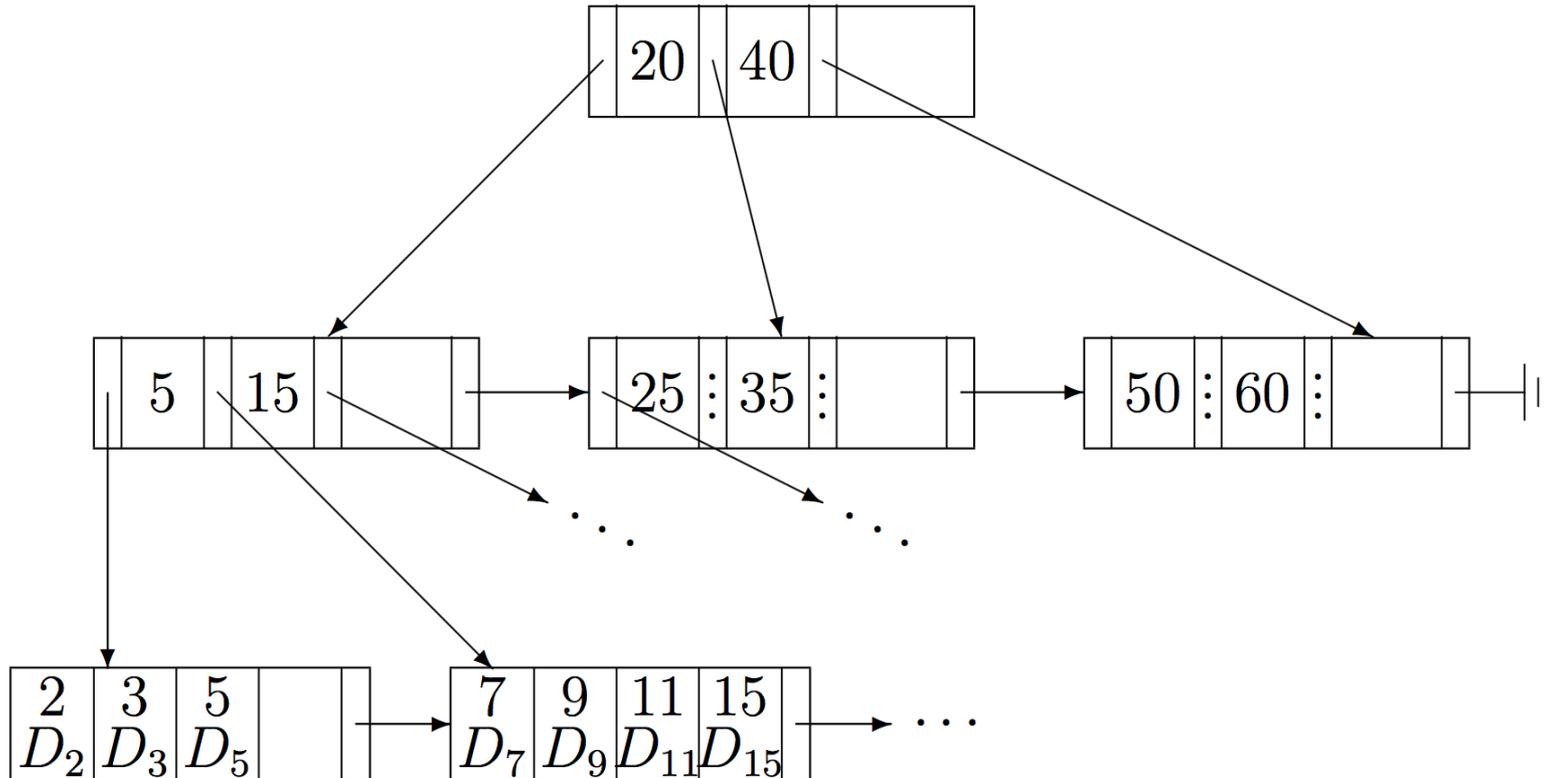


Es wäre theoretisch möglich, Indexstrukturen genauso wie „normale“ Daten zu behandeln. Dann würden die Datensätze eines Indexes – also z.B. Knoten eines B+- Baums – denselben Synchronisations- und Recoverytechniken unterliegen, wie die anderen Datensätze eines DBMS. Diese Vorgehensweise ist aber i.A. zu aufwendig für Indexstrukturen:

- Indices enthalten redundante, d.h. aus dem „normalen“ Datenbestand abgeleitete Informationen. Deshalb kann man abgeschwächte – und daher weniger aufwendige – Recoverytechniken einsetzen.
- Für die Mehrbenutzersynchronisation ist das Zweiphasen-Sperrprotokoll – das am häufigsten eingesetzte Synchronisationsverfahren für normale Datenbestände – zu aufwendig. Aus der speziellen Bedeutung der Indexeinträge lassen sich abgeschwächte Synchronisationstechniken konzipieren, die mehr Parallelität gewähren.

# Synchronisation von Indexstrukturen

## $B^+$ -Baum mit *rechts*-Verweisen zur Synchronisation



# Zusammenspiel Einfügen/Suchen

Wir wollen das verzahnte Zusammenspiel zweier Operationen auf dem B+-Baum aus obiger Abbildung illustrieren:

- Suchen(15)
- Einfügen(14)

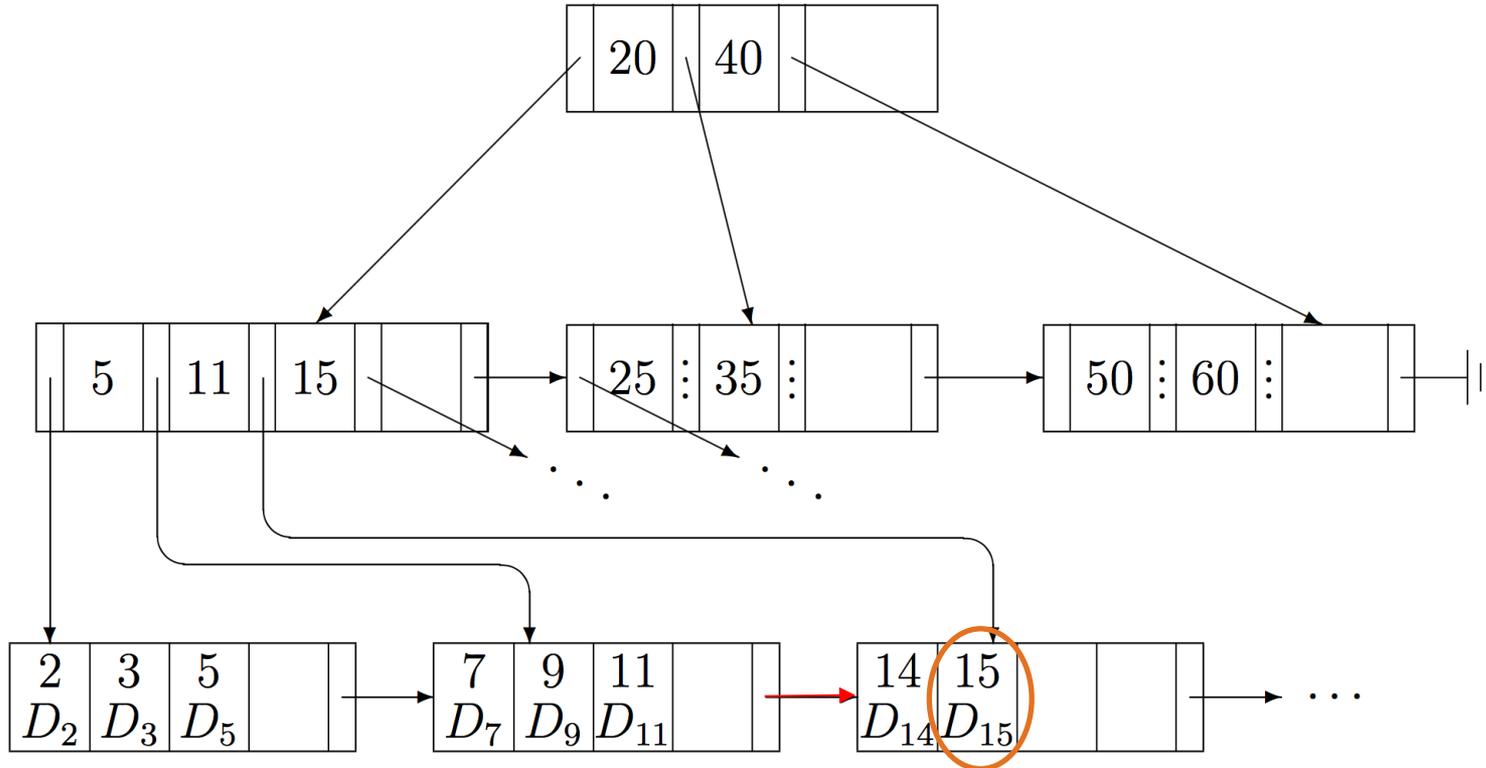
Wir nehmen an, dass die Suchoperation als erstes startet und die Wurzel und den linken Knoten der zweiten Stufe inspiziert. Als nächstes würde die Suche den zweiten Blattknoten von rechts besuchen. Jetzt nehmen wir aber an, dass zu diesem Zeitpunkt ein Kontextwechsel stattfindet, so dass Einfügen(14) ausgeführt wird. Im Zuge des Einfügevorgangs wird der zweite Blattknoten von rechts aufgespalten und der Zustand aus der nächsten Abbildung erzeugt. Wenn jetzt die Ausführung der Operation Suchen(15) wieder aufgenommen wird, befindet sich der Eintrag 15 nicht mehr auf der ursprünglich ermittelten Seite (2. von rechts). Deshalb muss die Suche auf den rechten Geschwisterknoten ausgedehnt werden.

## **Löschen von Einträgen**

Das Löschen von Einträgen ist bei diesem Verfahren der Synchronisation problematischer. Deshalb verzichtet man auf die Unterlaufbehandlung. Warum?

# Synchronisation von Indexstrukturen

**$B^+$ -Baum mit *rechts*-Verweisen nach Einfügen von 14**



# Transaktionsverwaltung in SQL92

## **set transaction**

[read only, |read write,]

### **[isolation level**

read uncommitted,  
|

read committed, |

repeatable read, |

serializable,]

**[diagnostic size ...,]**

# Transaktionsverwaltung in SQL92

**read uncommitted**: Dies ist die schwächste Konsistentstufe. Sie darf auch nur für **read only**-Transaktionen spezifiziert werden. Eine derartige Transaktion hat Zugriff auf noch nicht festgeschriebene Daten. Zum Beispiel ist folgender Schedule möglich:

$T_1$	$T_2$
	read(A)
	...
	write(A)
read(A)	
...	
	<b>rollback</b>

# Transaktionsverwaltung in SQL92

**read committed:** Diese Transaktionen lesen nur festgeschriebene Werte. Allerdings können sie unterschiedliche Zustände der Datenbasis-Objekte zu sehen bekommen:

$T_1$	$T_2$
read(A)	write(A) write(B)
	<b>commit</b>
read(B)	
read(A)	
...	

# Transaktionsverwaltung in SQL92

**repeatable read:** Das oben aufgeführte Problem des *non repeatable read* wird durch diese Konsistenzstufe ausgeschlossen. Allerdings kann es hierbei noch zum Phantomproblem kommen. Dies kann z.B. dann passieren, wenn eine parallele Änderungstransaktion dazu führt, dass Tupel ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.

**serializable:** Diese Konsistenzstufe fordert die Serialisierbarkeit. Dies ist der Default.