Synchronisation

- Konflikt-Kategorien
- Serialisierung
- Historien
- Sperrungen
- Verklemmungen
- Optimistische Synchronisation
- Synchronisation in SQL

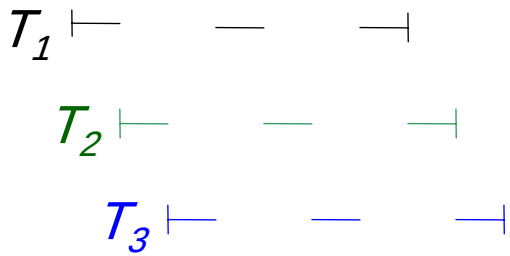
Kapitel 11

Mehrbenutzersynchronisation

Ausführung der drei Transaktionen T_1, T_2 und T_3:

(a) im Einzelbetrieb und

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



Fehler bei unkontrolliertem Mehrbenutzerbetrieb I

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)	

Fehler bei unkontrolliertem Mehrbenutzerbetrieb II

Abhängigkeit von nicht freigegebenen Änderungen

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

Serialisierbarkeit

- 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.	**BOT**	
2.	read(*A*)	
3.		**BOT**
4.		read(*C*)
5.	write(*A*)	
6.		write(*C*)
7.	read(*B*)	
8.	write(*B*)	
9.	**commit**	
10.		read(*A*)
11.		write(*A*)
12.		**commit**

Serielle Ausführung von T_1 vor T_2, also $T_1 \mid T_2$

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

Nicht serialisierbare Historie

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

Zwei verzahnte Überweisungs-Transaktionen

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
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.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

Ist nicht serialisierbar, obwohl dies im konkreten Fall nicht zum Konflikt führt. Letzteres kann die DB aber nicht beurteilen.

Eine Überweisung (T_1) und eine Zinsgutschrift (T_3)

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
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.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

Dieselbe r/w-Konstellation, diesmal zum Konflikt führend. Die DB greift zur Kontrolle nur auf die r/w-Struktur zu, nicht auf die Semantik der Anwendung!

Motivation: Zusammenhang ACID – Scheduler

- Um die Isolation (I in ACID) einer Transaktion zu gewährleisten, darf die Datenbank nur serialisierbare Historien zulassen.

- Der Scheduler stellt dies sicher, nicht indem er (im Nachhinein) schaut, ob die Historie serialisierbar war, sondern indem er nichtserialisierbare Historien gar nicht erst zulässt.

Theorie der Serialisierbarkeit

„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**.

Theorie der Serialisierbarkeit

Konsistenzanforderung einer Transaktion T_i

- entweder **abort** oder **commit** aber nicht beides!
- Falls T_i ein **abort** durchführt, müssen alle anderen Operationen $p_i(A)$ vor a_i ausgeführt werden, also $p_i(A) <_i a_i$.
- Analoges gilt für das **commit**, d.h. $p_i(A) <_i c_i$ 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 $w_i(A) <_i r_i(A)$.

Theorie der Serialisierbarkeit II

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_i 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.

Formale Definition einer Historie

Eine Historie ist eine partiell geordnete Menge (H, $<_H$) mit

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

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

$$\forall\, i\colon x <_{H_i} y \Rightarrow x <_H y$$

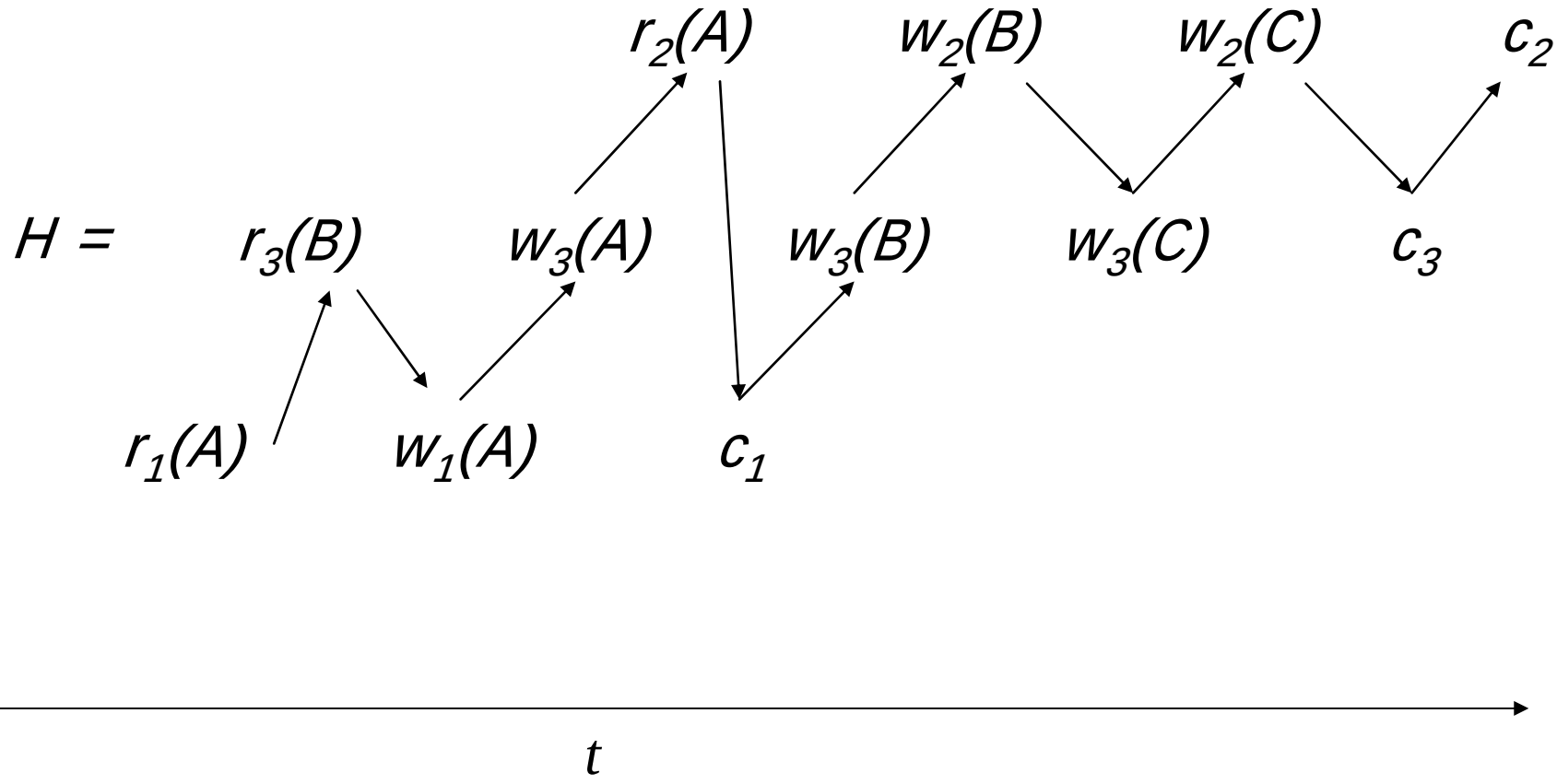
Die Ordnung muss nicht total sein, z.B. bei Mehrprozessorbetrieb.

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

  - $p <_H q$ oder

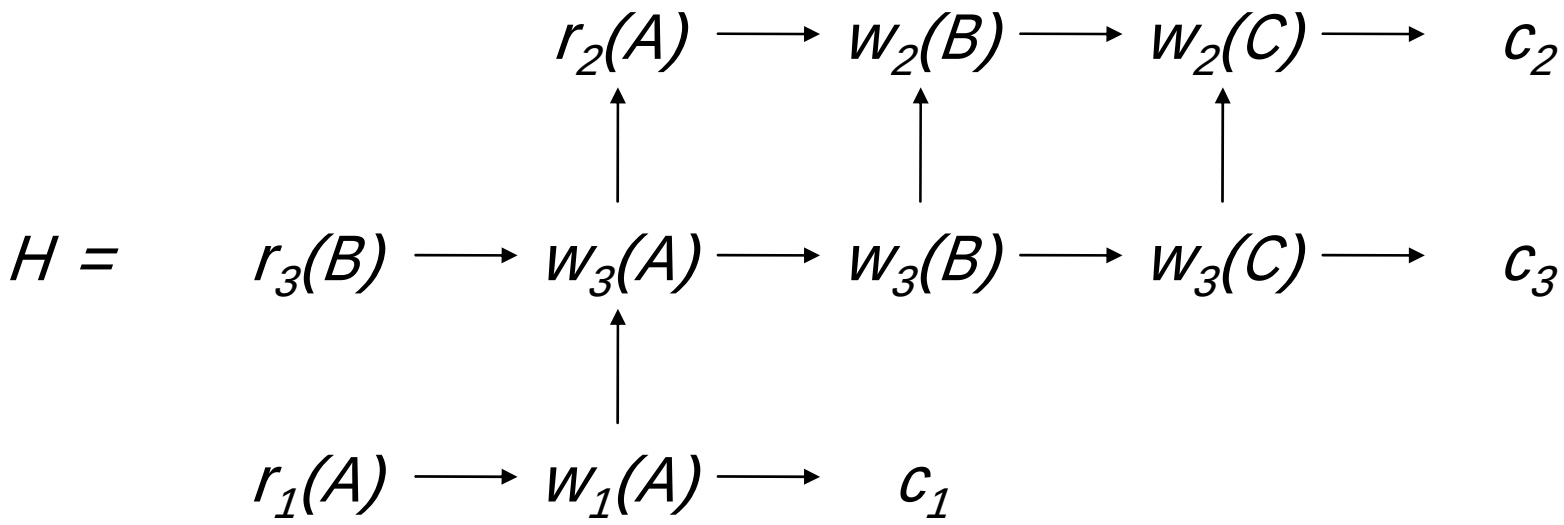
  - $q <_H p$.

Beispiel-Historie mit vollständig geordneter Zeit



Historie für drei Transaktionen

Beispiel-Historie für 3 TAs



Äquivalenz zweier Historien

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

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

(s. Folie 6)

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

(s. Folie 7)

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

(s. Folie 6)

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

(s. Folie 7)

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

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

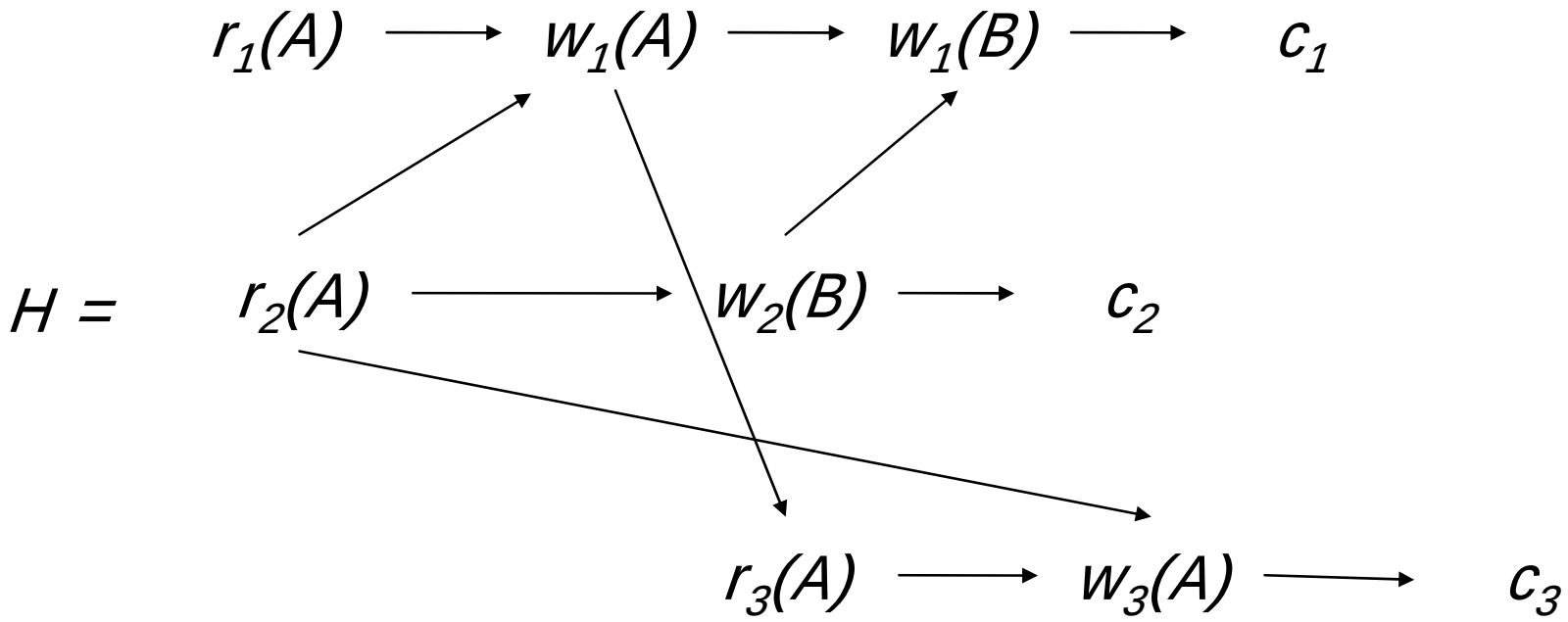
$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

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

Serialisierbare Historie

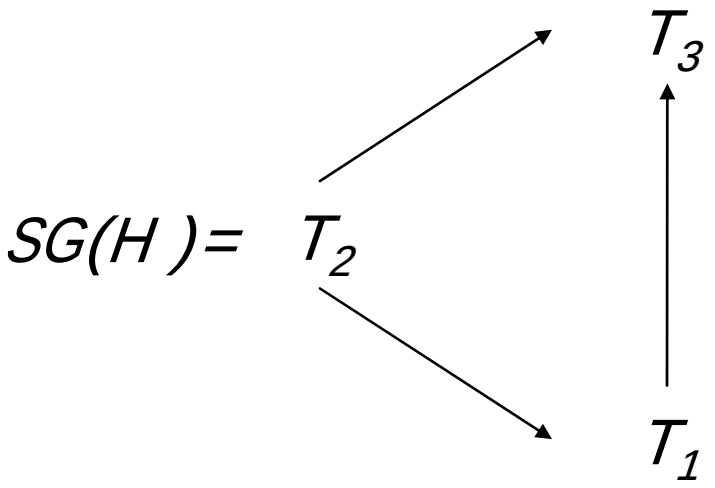
Eine Historie ist *serialisierbar,* wenn sie äquivalent zu einer seriellen Historie *Hs* ist.

Historie ...



Serialisierbarkeitsgraph

... 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

Serialisierbarkeitstheorem

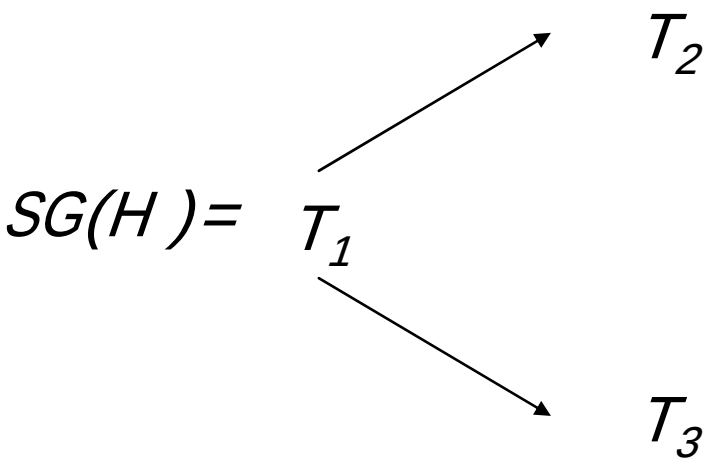
Satz: Eine Historie *H* ist genau dann *serialisierbar*, wenn der zugehörige Serialisierbarkeitsgraph *SG(H)* azyklisch ist.

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)

$H_s^1 = T_1 | T_2 | T_3$

$H_s^2 = T_1 | T_3 | T_2$

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

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(**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.

Strikte Historien

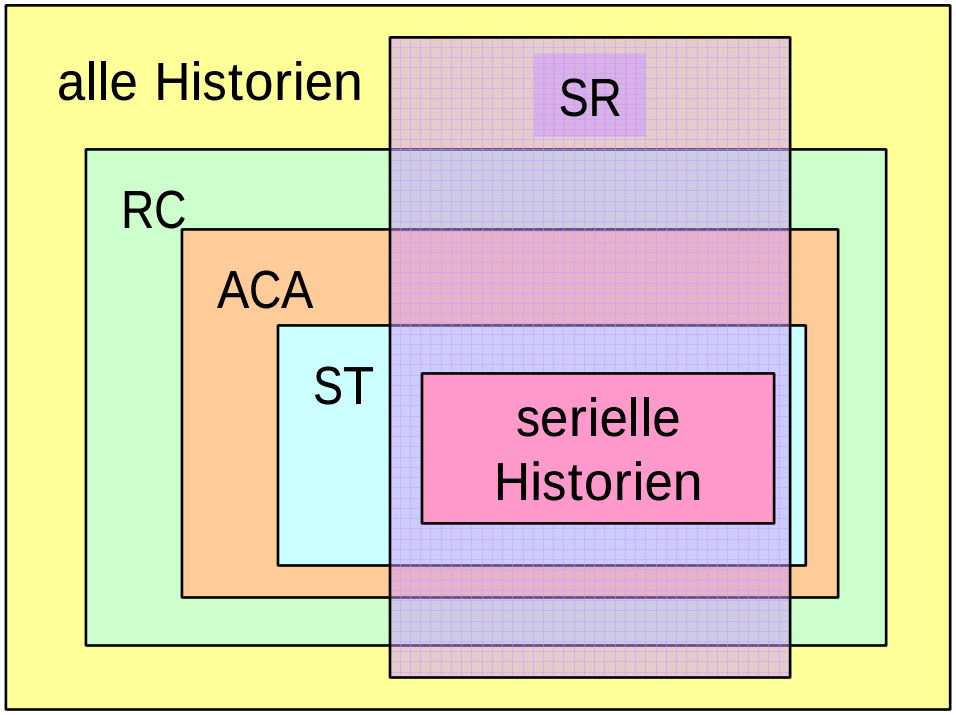
Eine Historie ist strikt, wenn für je zwei TAs T_i und T_j gilt:

Wenn $w_j(A) <_H o_i(A)$ *(mit* $o_i = w_i$ *oder* $o_i = r_i$*)* ,

dann muss entweder

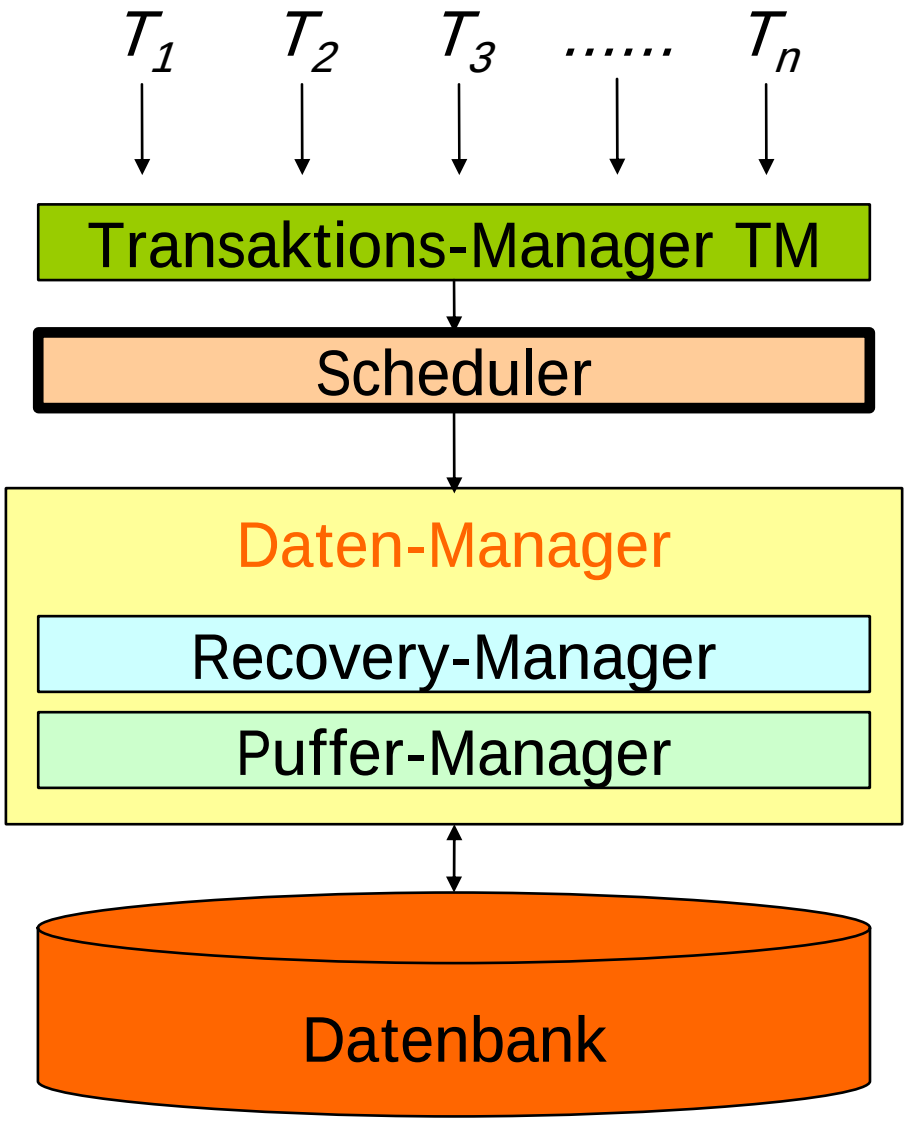
- $a_j <_H o_i(A)$ oder
- $c_j <_H o_i(A)$ gelten.

Beziehungen zwischen den Klassen von Historien



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

Der Datenbank-Scheduler



Aufgabe des Schedulers

- Reihung der Operationen verschiedener Transaktionen, so dass die Historie
  - mindestens serialisierbar
  - und meistens auch ohne kaskadierendes Rollback rücksetzbar ist.
- Die entstehenden Historien sollten also aus dem Bereich $ACA \cap SR$ stammen.
- Verschiedene Ansätze möglich:
  - sperrbasierte Synchronisation (wird am häufigsten verwandt)
  - Zeitstempel-basierte Synchronisation
  - optimistische Synchronisation (eignet sich bei vorwiegend lesenden Zugriffen)

Sperrbasierte Synchronisation

Zwei Sperrmodi

- *S* (shared, read lock, Lesesperre):
- *X* (exclusive, write lock, Schreibsperre):

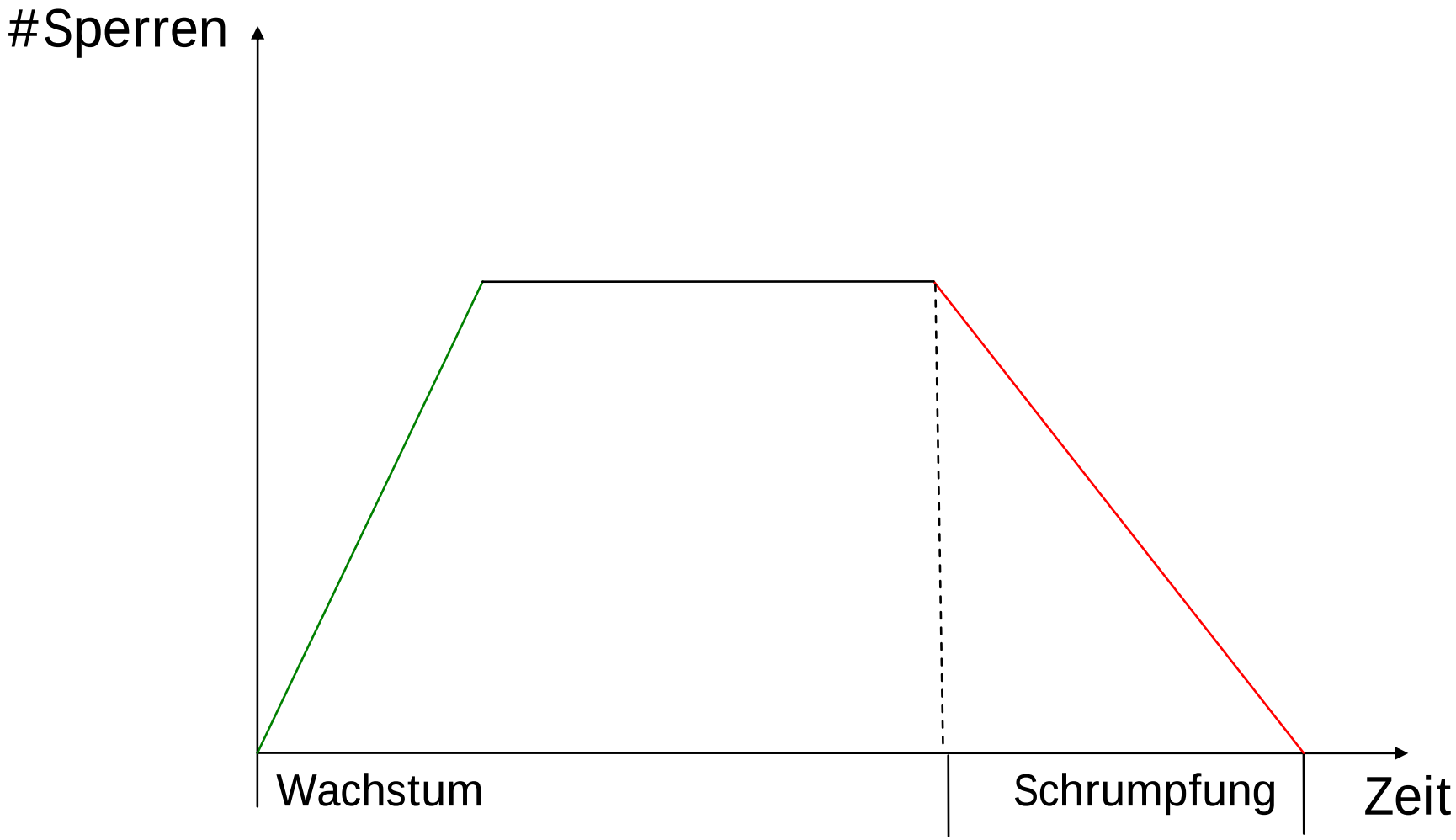
- *Verträglichkeitsmatrix* (auch *Kompatibilitätsmatrix* genannt)

	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

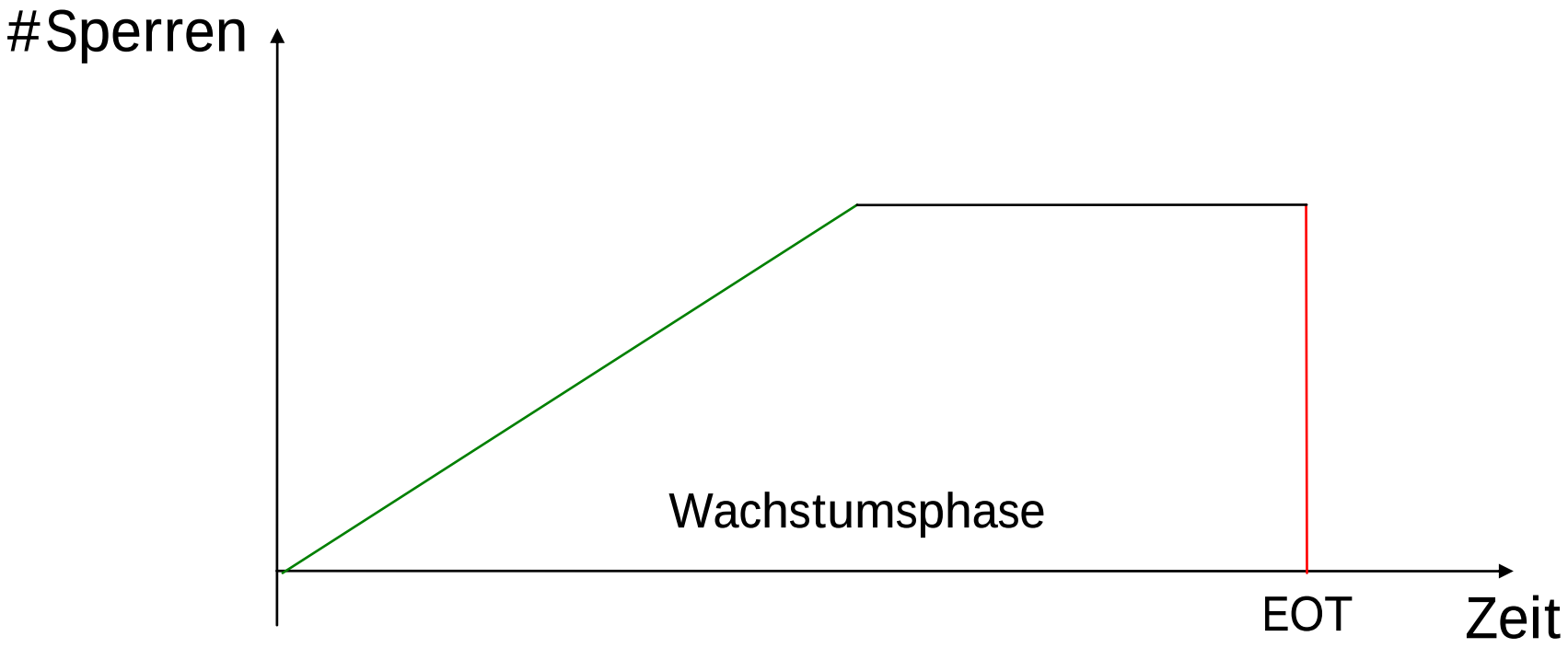
- T_1 modifiziert nacheinander die Datenobjekte A und B (z.B. eine Überweisung)
- T_2 liest nacheinander dieselben Datenobjekte A und B (z.B. zur Aufsummierung der beiden Kontostände).

Verzahnung zweier TAs gemäß 2PL

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

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.	**BOT**		
2.	**lockX**(*A*)		
3.		**BOT**	
4.		**lockS**(*B*)	
5.		read(*B*)	
6.	read(*A*)		
7.	write(*A*)		
8.	**lockX**(*B*)		T_1 muss warten auf T_2
9.		**lockS**(*A*)	T_2 muss warten auf T_1
10.	...	...	$\Rightarrow$ *Deadlock*

Verklemmungen

- **können entdeckt und dann aufgelöst**
- **oder gleich vermieden werden.**

Beides ist teuer. Mögliche Techniken:

Ad a:

1. **Time-Out**
2. **Zyklenerkennung**

Ad b:

3. **Preclaiming**
4. **Zeitstempel**

Erkennen von Verklemmungen

1. Brute-Force-Methode: Time-Out

- **Nach gewisser Wartezeit (z.B. 1 sec) wird Transaktion zurückgesetzt**
- **Nachteil: Falls Zeit zu kurz, werden zu viele Transaktionen zurückgesetzt, die nur auf Resourcen (CPU etc.) warten. Falls Zeit zu lang, werden zu viele Verklemmungen geduldet.**

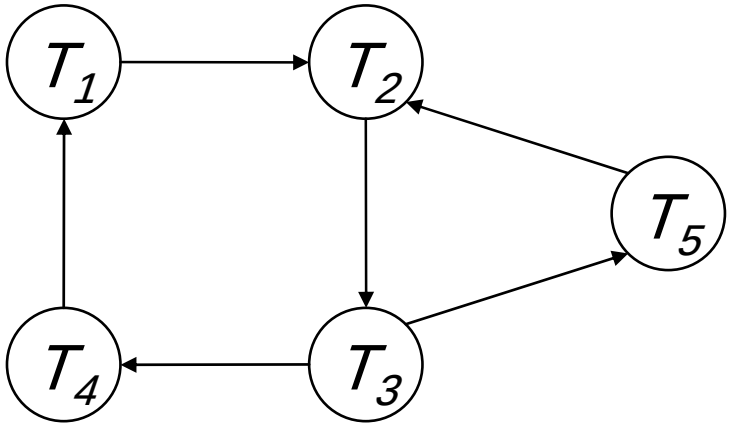
Erkennen von Verklemmungen

2. Zyklenerkennung durch Tiefensuche im Wartegraph

- ist aufwendiger, aber präziser

Bsp.: Wartegraph mit zwei Zyklen:

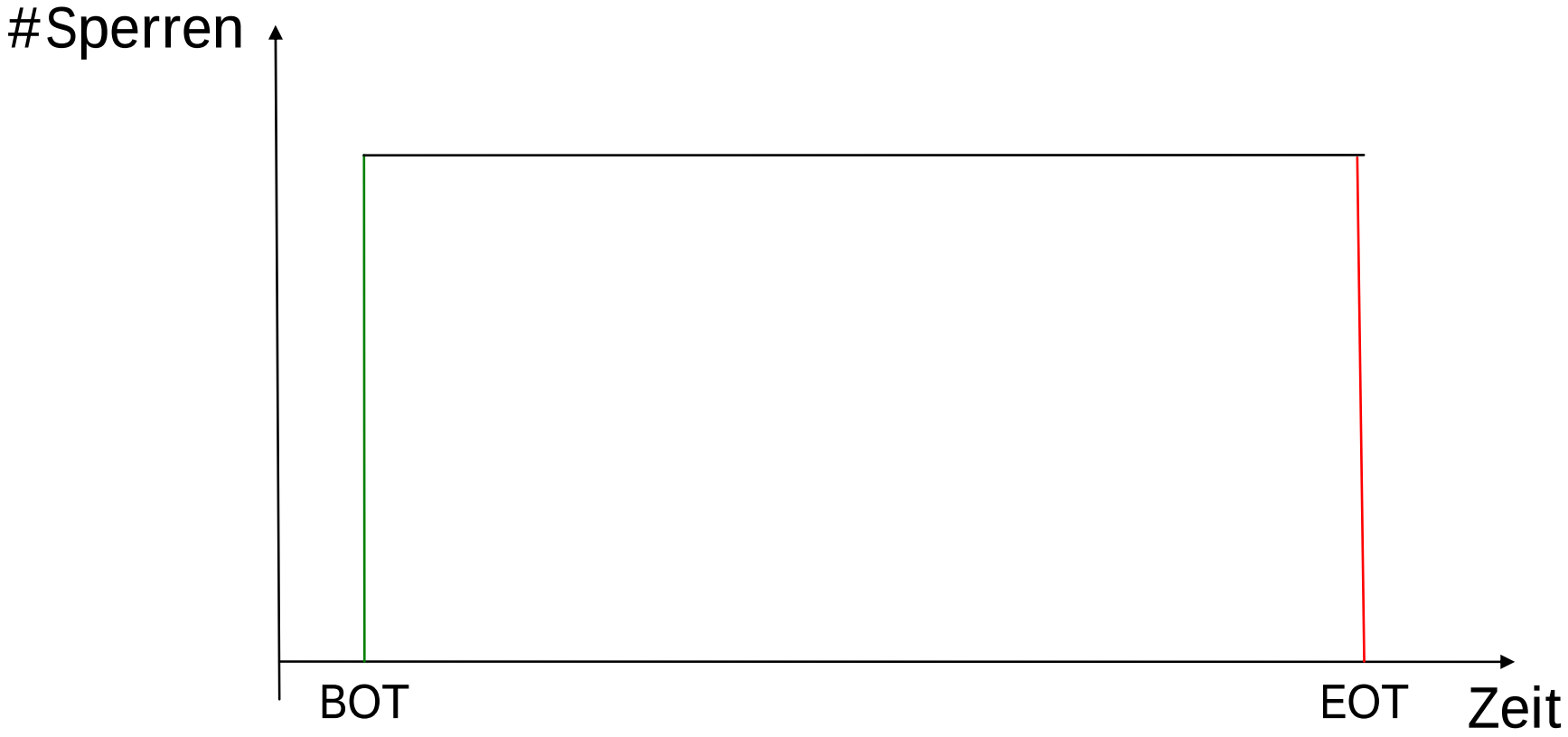
- $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$
- $T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$



- Beide Zyklen können durch Rücksetzen von T_3 „gelöst“ werden.

Vermeiden von Verklemmungen

3. Preclaiming in Verbindung mit dem strengen 2 PL-Protokoll



Preclaiming

ist in der Praxis häufig nicht einzusetzen:

- Man weiss i.A. apriori nicht genau, welche Sperren benötigt werden, z.B. bei if...then...else im Anwendungsprogramm.
- Daher müssen normalerweise mehr Sperren als nötig „auf Verdacht“ reserviert werden, was zu übermäßiger Resourcenbelegung und Einschränkung der Parallelität führt.

Vermeiden von Verklemmungen

4. 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.
- Zwei Varianten:

wound-wait Strategie

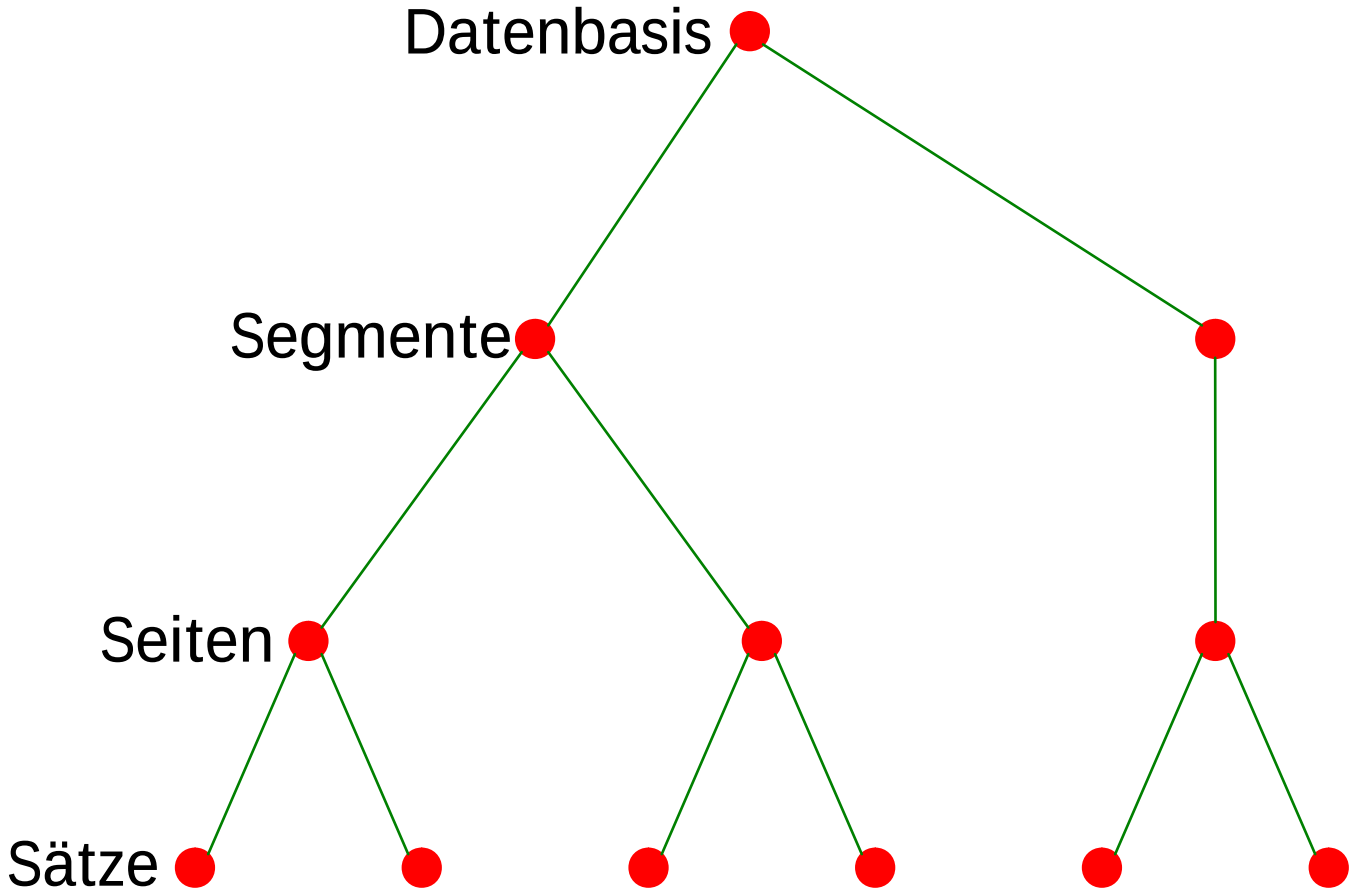
- T_1 will Sperre erwerben, die von T_2 gehalten wird.
- Wenn T_1 älter als T_2 ist, wird T_2 abgebrochen und zurückgesetzt, so dass T_1 weiterlaufen kann.
- Sonst wartet T_1 auf die Freigabe der Sperre durch T_2.

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.

MGL: Multi-Granularity Locking

Hierarchische Anordnung möglicher Sperrgranulate



MGL: Multi-Granularity Locking

- Bisher haben wir Sperrungen auf derselben Granularitätsebene betrachtet.
- Nachteil bei kleiner Granularität: Bei TAs mit Zugriff auf viele Tupel muss viel Sperraufwand betrieben werden.
- Nachteil bei großer Granularität: unnötige Sperrung von Datensätzen

- 1. Lösungsansatz:
  - Sperrung auf verschiedenen Hierarchieebenen erlaubt.
  - Bei Anforderung einer Sperre muss man überprüfen, ob weiter oben oder unten bereits Sperren gesetzt sind.

  → zu hoher Suchaufwand!
- 2. Lösungsansatz: Einführung zusätzlicher Sperrmodi → MGL

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.

Multi-Granularity Locking (MGL)

Kompatibilitätsmatrix

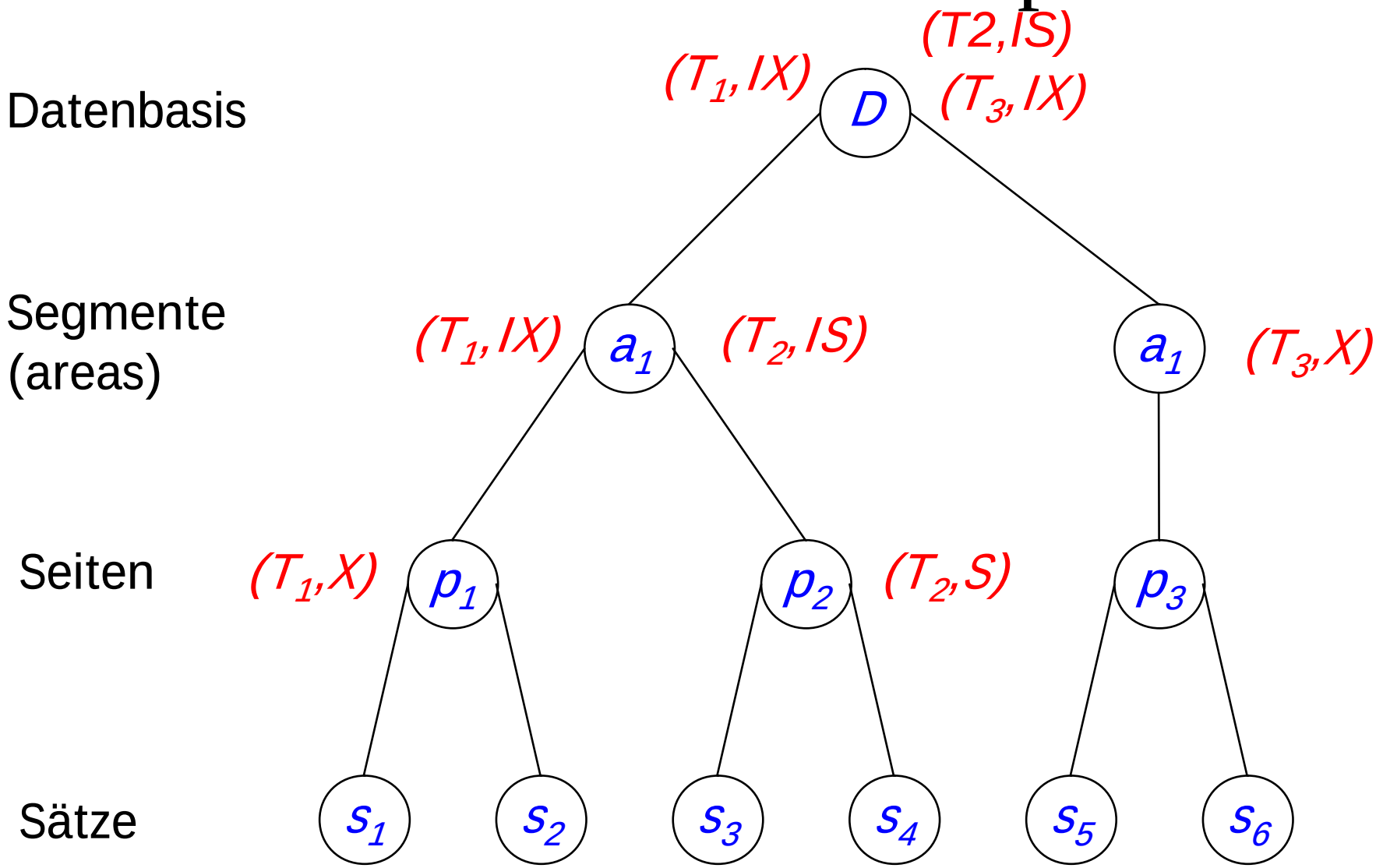
	NL	*S*	*X*	*IS*	*IX*
S	✓	✓	-	✓	-
X	✓	-	-	-	-
IS	✓	✓	-	✓	✓
IX	✓	-	-	✓	✓

Multi-Granularity Locking (MGL)

Sperrprotokoll des MGL

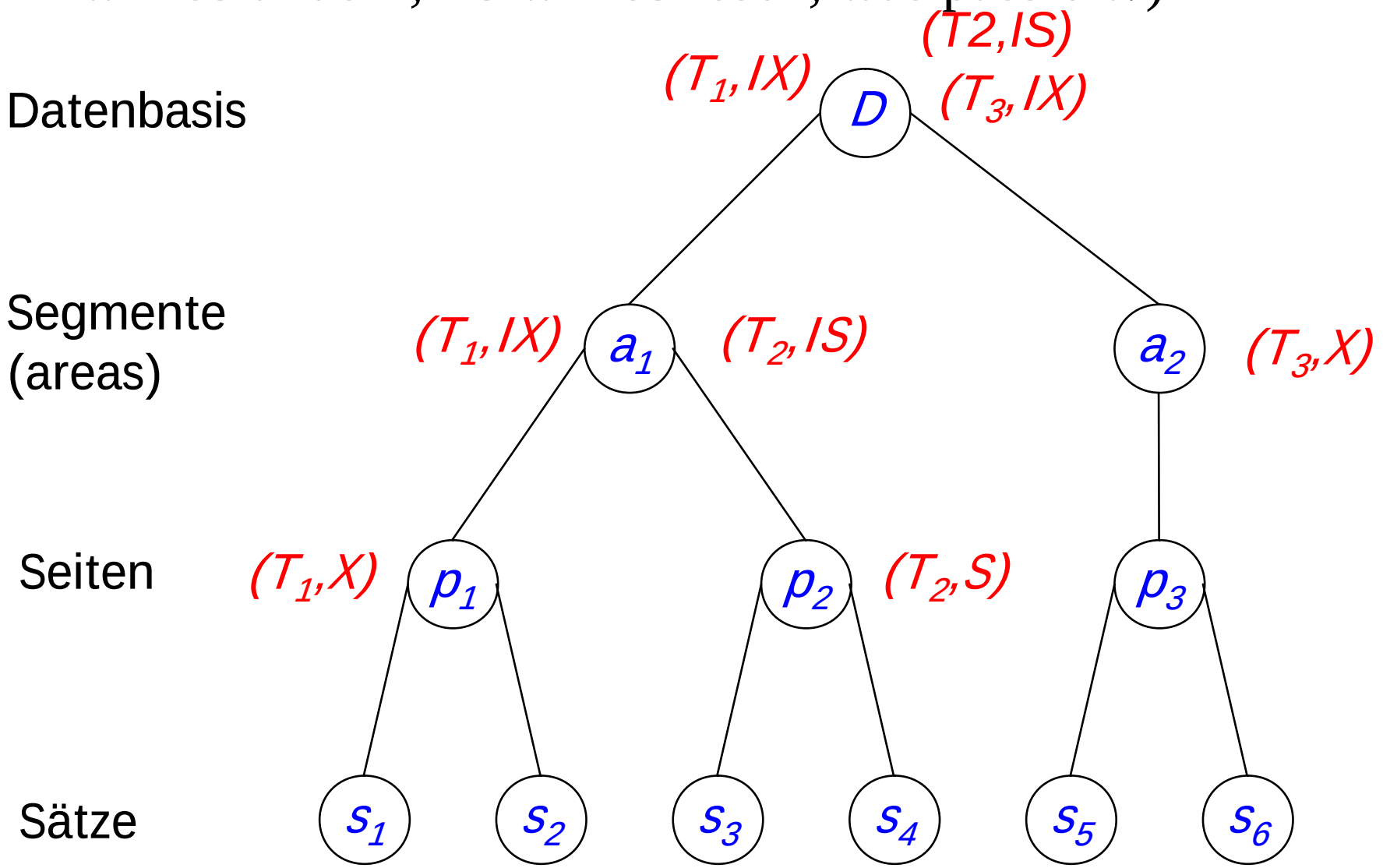
1. Bevor ein Knoten mit *S* oder *IS* gesperrt wird, müssen alle Vorgänger in der Hierarchie vom Sperrer (also der Transaktion, die die Sperre anfordert) im *IX*- oder *IS*-Modus gehalten werden.
2. Bevor ein Knoten mit *X* oder *IX* gesperrt wird, müssen alle Vorgänger vom Sperrer im *IX*-Modus gehalten 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

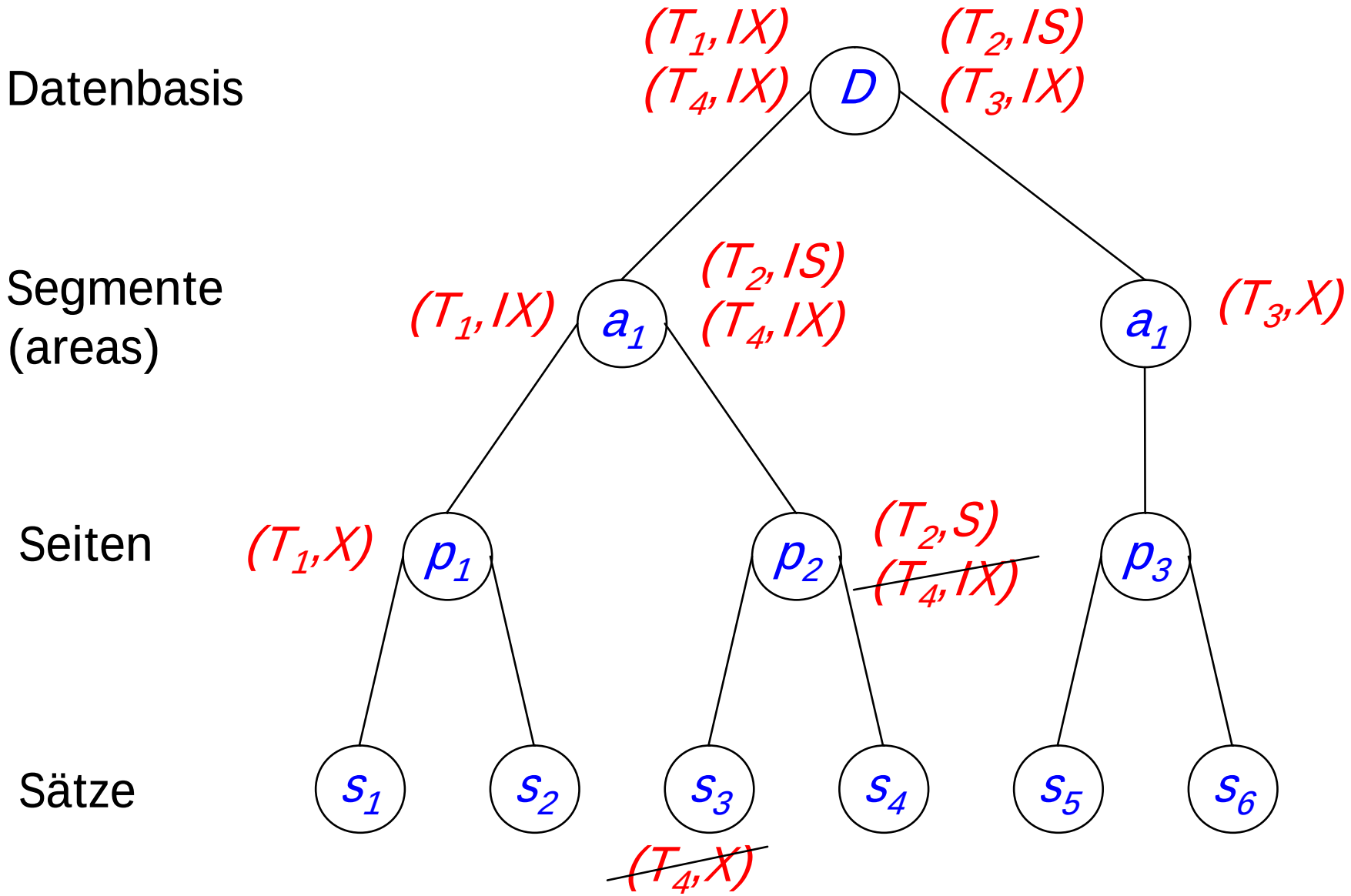


Datenbasis-Hierarchie mit Sperren

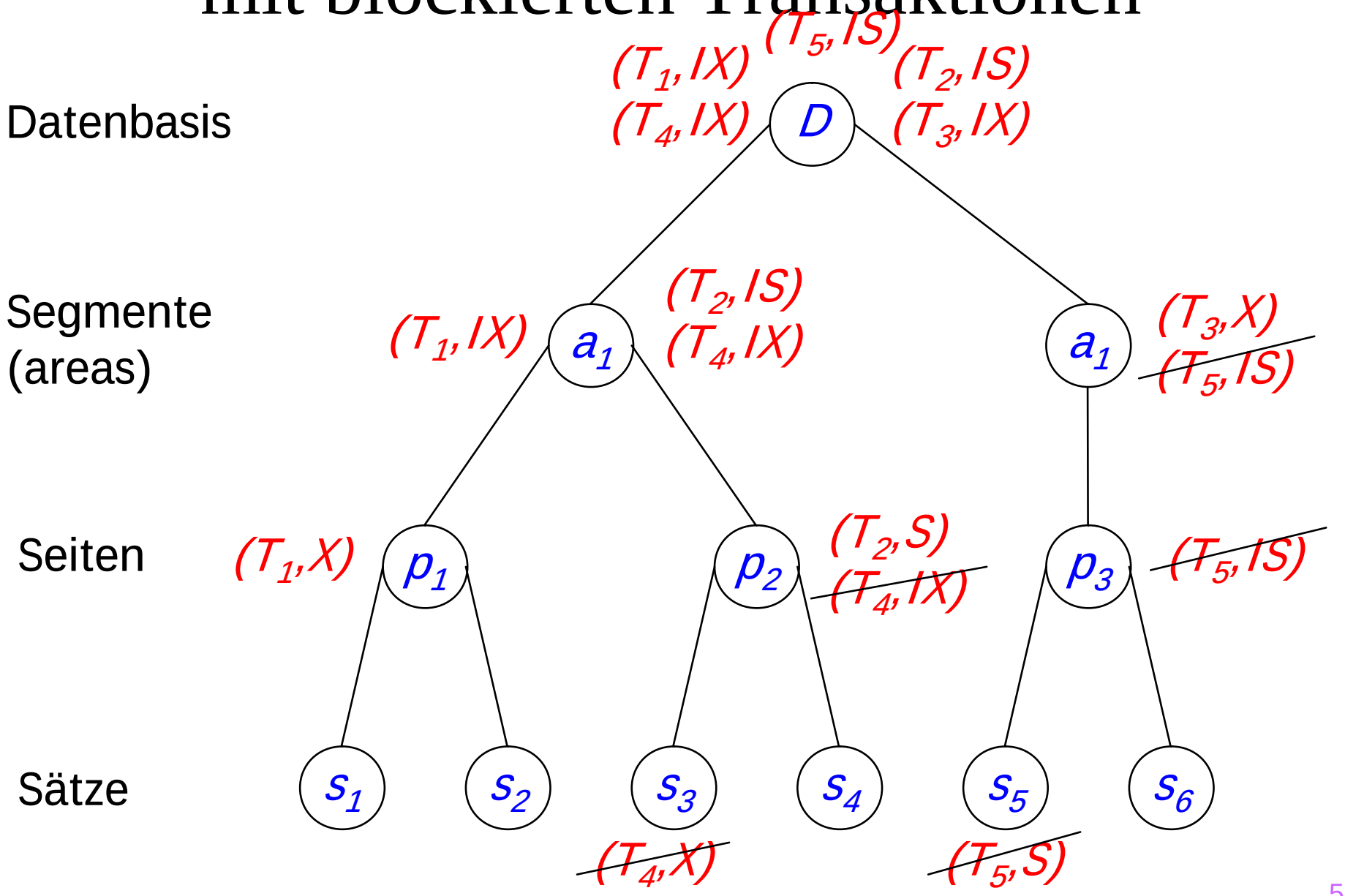
(T4 will s3 ändern, T5 will s5 lesen, was passiert?)



Datenbasis-Hierarchie mit blockierten Transaktionen



Datenbasis-Hierarchie mit blockierten Transaktionen



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
select count(*)	
from prüfen	
where Note **between** 1 **and** 2;	
	insert into prüfen
	values(19555, 5001, 2137, 1);
select count(*)	
from prüfen	
where Note **between** 1 **and** 2	

Phantomprobleme können immer noch auftreten, hier z.B. wenn die Sperren auf Satz-Ebene gesetzt werden.

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.

In dieser Fassung sind alle Historien serialisierbar, schließen aber kaskadierendes Rücksetzen nicht aus. → Protokoll muss geeignet erweitert werden.

Optimistische Synchronisation

1. *Lesephase:*
   - In dieser Phase werden alle Operationen der Transaktion ausgeführt – also auch die Änderungsoperationen.
   - Die Transaktion liest (zunächst) nur, und führt alle Schreiboperationen auf lokalen Variablen aus.

2. *Validierungsphase:*
   - 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. *Schreibphase:*
   - 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 gilt:

1. T_a war zum Beginn der Transaktion T_j schon abgeschlossen – einschließlich der Schreibphase.

1. 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:*

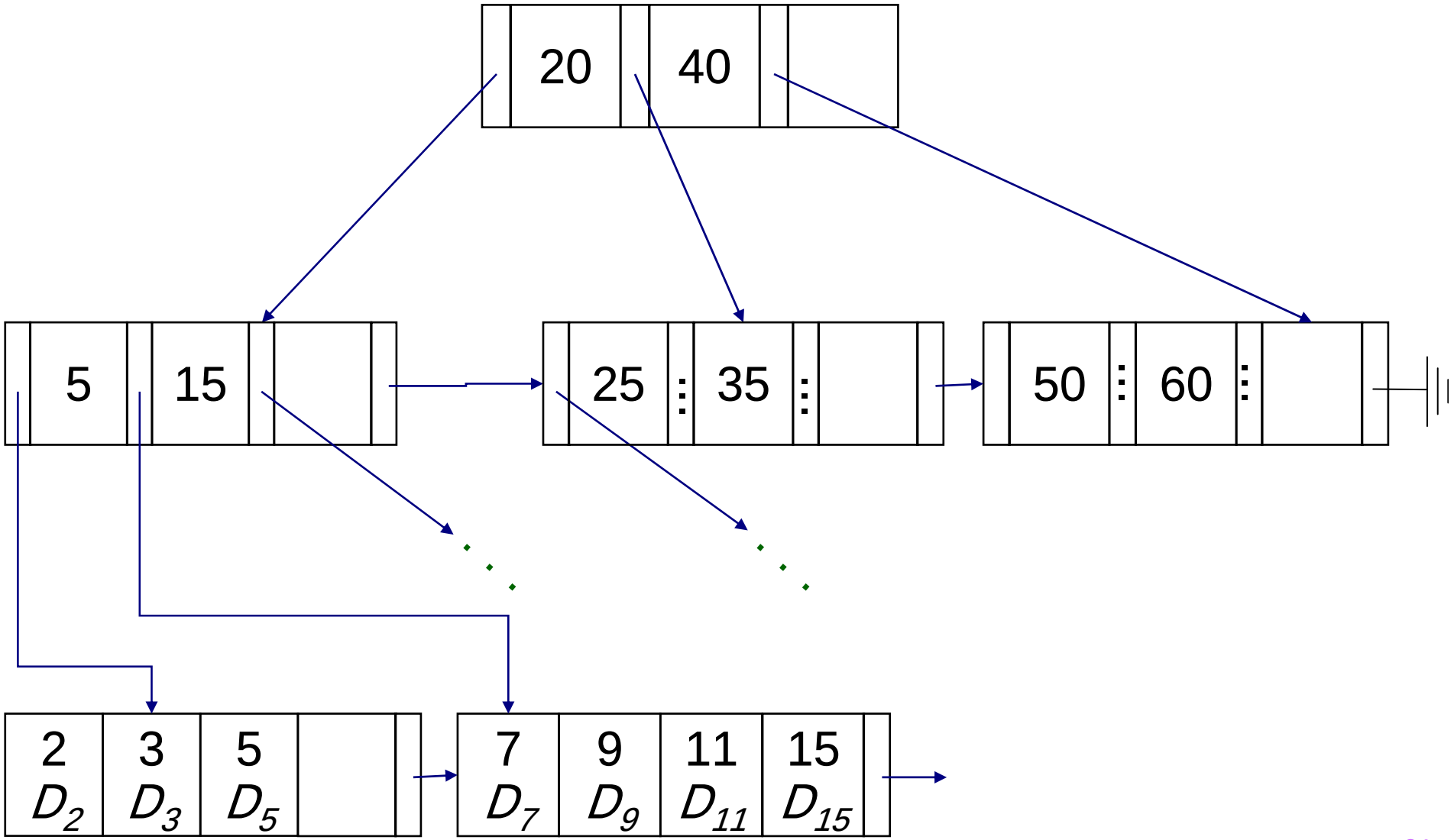
$$WriteSet(T_a) \cap ReadSet(T_j) = \varnothing$$

Synchronisation von Indexstrukturen

- Indexe enthalten redundante Informationen, deswegen kann man weniger aufwendige Recoverytechniken einsetzen.
- Für Indexe ist das Zweiphasen-Sperrprotokoll zu aufwändig. Bsp.:
  - Das strenge 2PL würde bei Lesezugriff auf B^+-Baum[1] jeden Knoten des Baumes mit Lesesperre versehen, weite Bereiche sind dann für Einfügeoperationen gesperrt.
- Lösung: kurze Lesesperre nur auf Wurzel, bis Eintrag gefunden. Dann Freigabe und Zugriff auf Daten. Wenn Datum dort nicht mehr gefunden wird (wegen Einfügeoperation), muss es weiter rechts liegen.

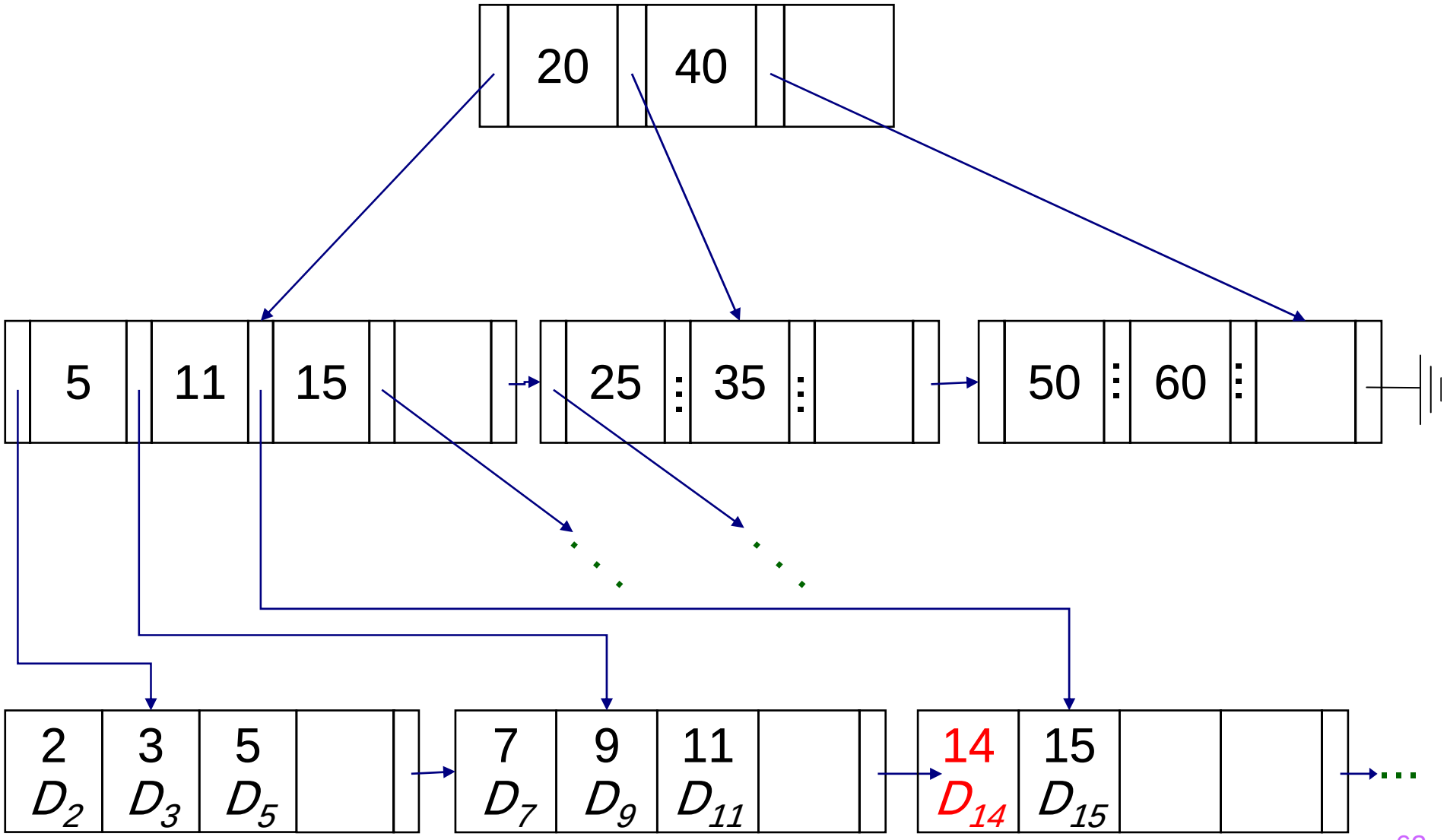
Synchronisation von Indexstrukturen

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



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)	
. . .	
	rollback

- Macht nur Sinn zum Browsen der Datenbank. Hindert andere Transaktionen nicht, da keine Sperren gesetzt werden.

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)
	commit
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 (in den meisten DBMS).