Mehrbenutzersynchronisation

VL Datenbanksysteme

Ingo Feinerer

Arbeitsbereich Datenbanken und Artificial Intelligence
Institut für Informationssysteme
Technische Universität Wien

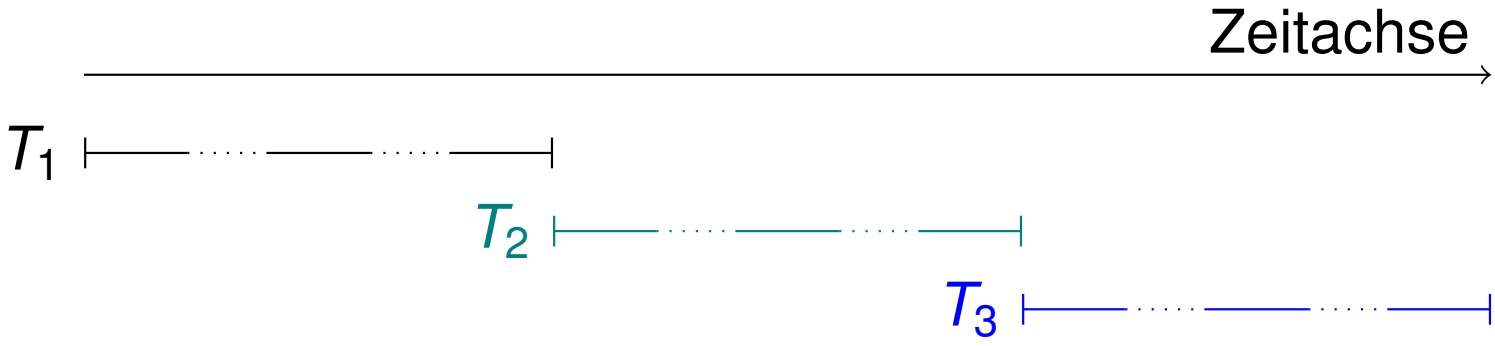
Nebenläufigkeit und mögliche Fehler

- Vorteil der verzahnten Ausführung
- Mögliche Konflikte

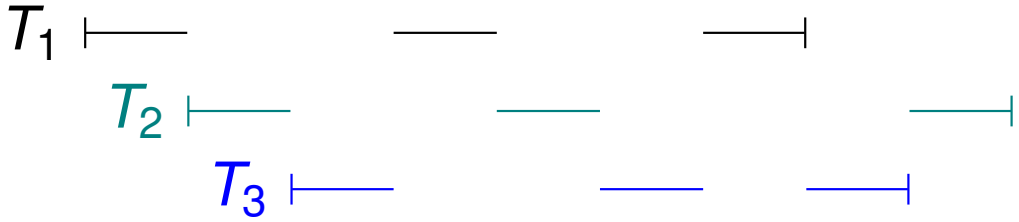
Nebenläufigkeit

Ausführung der drei Transaktionen T_1, T_2 und T_3:

(a) im Einbenutzerbetrieb



(b) im (verzahnten) Mehrbenutzerbetrieb



Verzahnte Ausführung

Idee:

- CPU- und I/O-Aktivitäten können parallel geschehen.
- Verzahnte Ausführung mehrerer Transaktionen führt zu besserer Auslastung dieser beiden Ressourcen.

Vorteile der verzahnten Ausführung:

- Durch verzahnte Ausführung kann der Durchsatz des DBMS erhöht werden (d.h. durchschnittliche Anzahl der abgeschlossenen Transaktionen pro Zeiteinheit).
- Unvorhersehbare Verzögerungen der Antwortzeit lassen sich dadurch reduzieren (z.B. bei serieller Ausführung muss eventuell eine kleine Transaktion hinter einer großen Transaktion sehr lange warten).

Konflikte und mögliche Fehler

Konflikte

- Immer wenn 2 Transaktionen auf dasselbe Objekt zugreifen und mindestens ein Zugriff schreibend erfolgt.
- Mögliche Konflikte: W-W, W-R, R-W

Mögliche Fehler:

- Lost Update (W-W)
- Dirty Read (W-R)
- Unrepeatable Read (R-W)
- Phantomproblem (R-W bei Insert-Operation)

Verlorengegangene Änderungen

Lost Update

Beispiel

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)	

Problem: Bei commit von T_1 geht die Änderung von T_2 verloren.

Lesen nicht freigegebener Änderungen

Dirty Read

Beispiel

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**	

Problem: Die Änderungen, die T_2 durchführt, gehen von einem inkonsistenten DB-Zustand aus.

Überschreiben von Daten, die noch gelesen werden

Unrepeatable Read

Beispiel

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

Problem: Wiederholtes Lesen durch T_1 liefert unterschiedliche Ergebnisse, obwohl T_1 keine Änderung vorgenommen hat.

Phantomproblem

Unrepeatable Read

Beispiel

Schritt	T_1	T_2
1.		**SELECT SUM**(KontoStand) **FROM** Konten
2.	**INSERT INTO** Konten **VALUES** (C,1000,...)	
3.		**SELECT SUM**(KontoStand) **FROM** Konten

Problem: Die Transaktion T_2 berechnet unterschiedliche Werte, da in der Zwischenzeit das „Phantom“ mit den Werten $(C, 1000, \ldots)$ eingefügt wurde.

Serialisierbarkeit

- Historie (Schedule)
- Serialisierbare vs. nicht serialisierbare Historien
- Formale Definition der Serialisierbarkeit

Historie (Schedule)

Elementare 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 **abort**
- c_i zur Durchführung des **commit**
- (insert und delete werden vorerst nicht betrachtet)

Definition

Unter einer *Historie (Schedule)* versteht man die zeitliche Anordnung der elementaren Operationen von einer Menge von Transaktionen.

Die Ordnung der elementaren Operationen innerhalb einer Transaktion muss dabei erhalten bleiben.

Serialisierbarkeit

Serielle Historie:

- Jede Transaktion wird vollständig abgearbeitet, bevor die nächste beginnt.
- Schreibweise: $T_1 \mid T_2 \mid T_3 \ldots$: „T_1 vor T_2 vor T_3 …“.

Serialisierbarkeit (informell):

- Eine (verzahnte) Historie heißt „serialisierbar“, wenn sie — aus der Sicht des DBMS — denselben Effekt hat, wie eine serielle Ausführung.
- Das DBMS sieht dabei nur die Elementaroperationen **read**, **write**, etc. aber keine Anwendungslogik.

Serialisierbare Historie

Beispiel

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**

Äquivalente serielle Ausführung: $T_1 \mid T_2$

Beispiel

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

Beispiel

Schritt	T_1	T_2
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**	

Bemerkung

- Aus der Sicht des DBMS ist diese Historie nicht serialisierbar (z.B. sowohl $T_1 \mid T_3$ als auch $T_3 \mid T_1$ würde zu lost updates führen).
- Wegen spezieller Anwendungslogik kann es trotzdem sein, dass 2 Transaktionen, die zu dieser Historie passen, denselben Effekt wie eine serielle Historie haben (siehe nächstes Beispiel).
- Es kann für diese Historie aber auch 2 verzahnte Transaktionen geben, die nicht denselben Effekt wie eine serielle Historie haben (siehe übernächstes Beispiel).

Zwei Überweisungs-Transaktionen

Beispiel

Schritt	T_1	T_2
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**	

Eine Überweisung und eine Zinsgutschrift

Beispiel

Schritt	T_1	T_2
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**	

Formale Definition einer Transaktion

Eine Transaktion T_i ist charakterisiert durch:

- Angabe der Elementaroperationen $r_i(A)$, $w_i(A)$, a_i und c_i
- Angabe einer (partiellen) Ordnung $<_i$ zwischen diesen Operationen

Konsistenzanforderungen:

- entweder abort oder commit aber nicht beides
- falls T_i ein abort bzw. commit durchführt, müssen alle anderen Operationen vor a_i bzw. c_i ausgeführt werden, d.h. $r_i(A) <_i a_i$, $w_i(A) <_i a_i$, bzw. $r_i(A) <_i c_i$, $w_i(A) <_i c_i$.
- 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 $w_i(A) <_i r_i(A)$.

Formale Definition einer Historie

Eine Historie H ist charakterisiert durch:

- Menge von Transaktionen $\{T_1, T_2, \ldots, T_n\}$
- (partielle) Ordnung $<_H$ zwischen den Elementaroperationen dieser Transaktionen

Konsistenzanforderungen:

- $<_H$ ist verträglich mit allen $<_i$ Ordnungen, d.h.: $<_H \supseteq \bigcup_{i=1}^{n} <_i$
- Für zwei Konfliktoperationen $r_i(A)$, $w_j(A)$ bzw. $w_i(A)$, $w_j(A)$ ist die Ordnung in H festgelegt, d.h.:
  - für alle $r_i(A)$, $w_j(A)$ gilt $r_i(A) <_H w_j(A)$ oder $w_j(A) <_H r_i(A)$
  - für alle $w_i(A)$, $w_j(A)$ gilt $w_i(A) <_H w_j(A)$ oder $w_j(A) <_H w_i(A)$

Historie für drei Transaktionen

Beispiel

$$H = \begin{array}{ccccccc} & & r_2(A) & \longrightarrow & w_2(B) & \longrightarrow & w_2(C) \longrightarrow c_2 \\ & & \uparrow & & \uparrow & & \uparrow \\ r_3(B) & \longrightarrow & w_3(A) & \longrightarrow & w_3(B) & \longrightarrow & w_3(C) \longrightarrow c_3 \\ & & \uparrow & & & & \\ r_1(A) & \longrightarrow & w_1(A) & \longrightarrow & c_1 & & \end{array}$$

Äquivalenz zweier Historien

Definition

Zwei Historien H und H' heißen *äquivalent* (Schreibweise: $H \equiv H'$), wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen.

Beispiel

Äquivalente Historien:

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

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

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

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

Serialisierbare Historie

Definition

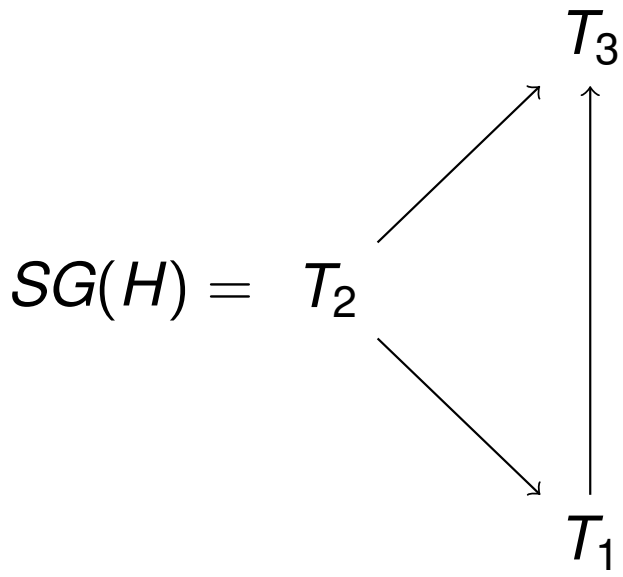
Eine Historie ist *serialisierbar*, wenn sie äquivalent zu einer seriellen Historie ist.

Beispiel

Serialisierbare Historie:

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

Serialisierbarkeitsgraph



- Knoten von $SG(H)$: Transaktionen T_1, T_2, ... von H
- Gerichtete Kanten $T_i \to T_j$ in $SG(H)$, falls für ein Paar von Konfliktoperationen p_i, p_j gilt: $p_i <_H q_j$

Beispiel

$w_1(A) \to r_3(A)$ von H ergibt Kante $T_1 \to T_3$ in $SG(H)$.

Serialisierbarkeitstheorem

Theorem

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

Definition

Eine *topologische Sortierung* von H ist eine serielle Anordnung der TAs in H, konsistent mit den gerichteten Kanten in $SG(H)$.

Beispiel

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

$$SG(H) = \begin{array}{l} T_1 \nearrow T_2 \\ T_1 \searrow T_3 \end{array}$$

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

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

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

Berechnung einer topologischen Sortierung

- Beobachtung: In einem azyklischen, gerichteten Graphen gibt es mindestens einen Knoten ohne eingehende Kante.
- Algorithmus:

  In: azyklischer Serialisierbarkeitsgraph G
  Out: eine mögliche topologische Sortierung

  1: Wähle in G einen Knoten N ohne eingehende Kante
  2: Schreibe Label T_i von N in den Output
  3: Lösche aus G den Knoten N und alle von N ausgehenden Kanten
  4: **if** noch ein Knoten übrig ist **then**
  5: **goto** 1
  6: **end if**

Weitere Eigenschaften von Historien

- Rücksetzbare Historien
- Kaskadierendes Rücksetzen
- Strikte Historien

Rücksetzbare Historien

Definition

Transaktion T_i *liest von* T_j in der Historie H, wenn die folgenden Bedingungen gelten:

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.

Idee: Transaktion T_i liest also das Datum A genau mit dem Wert, den T_j geschrieben hat.

Rücksetzbare Historien

Definition

Eine Historie H heißt *rücksetzbar*, wenn für alle Transaktionen T_i und T_j in H gilt: Falls T_i von T_j liest, dann darf T_i nicht vor T_j das commit durchführen.

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

Definition

Eine Historie H *vermeidet kaskadierendes Rücksetzen*, wenn für alle Transaktionen T_i und T_j in H gilt: T_i liest von T_j ein Datum A erst, nachdem T_j committed wurde, d.h. $c_j <_H r_i(A)$.

Historie mit kaskadierendem Rücksetzen

Beispiel

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**)				

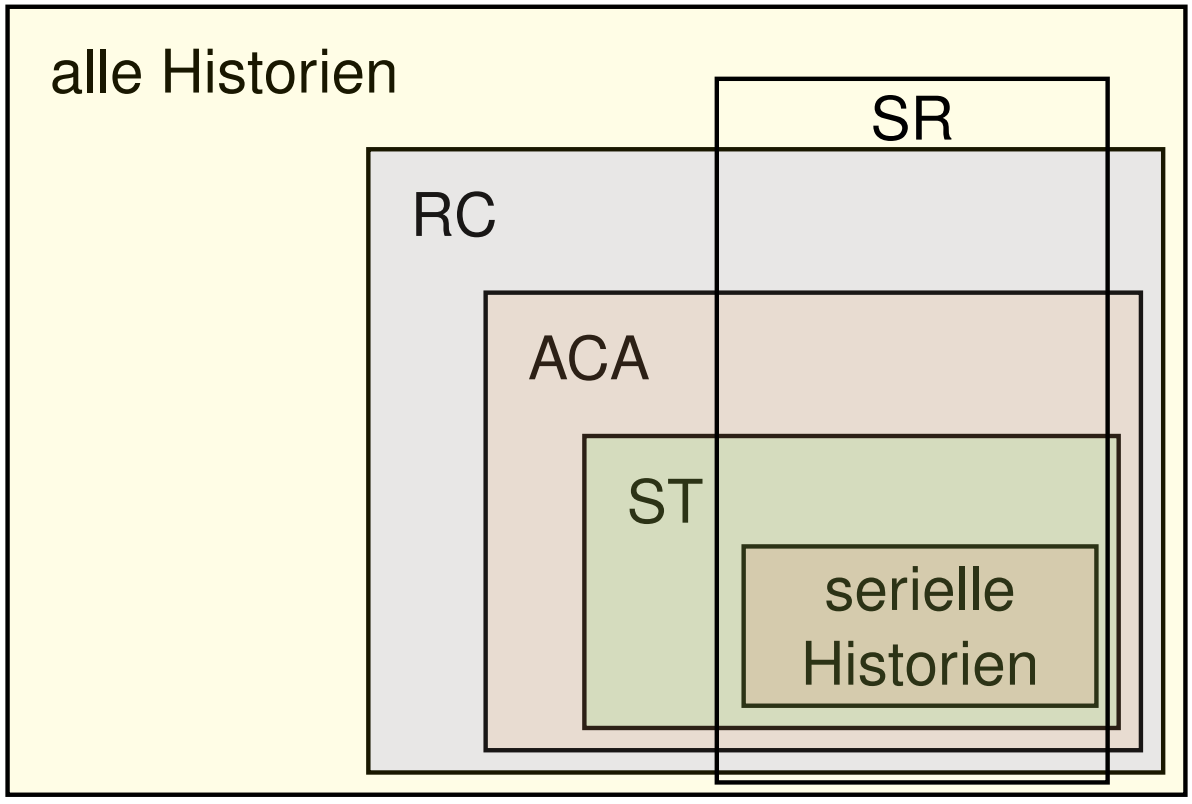
Strikte Historien

Definition

Eine Historie H heißt *strikt*, wenn für alle Transaktionen T_i und T_j in H gilt: falls $w_j(A) <_H o_i(A)$ mit $o_i = r_i$ oder $o_i = w_i$ dann gilt entweder $a_j <_H o_i(A)$ oder $c_j <_H o_i(A)$.

Anders ausgedrückt: Falls ein Datum A von einer Transaktion T_i geschrieben wurde, dürfen andere Transaktionen erst nach der Beendigung von T_i (mit commit oder abort) auf A zugreifen.

Beziehungen zwischen den Klassen von Historien



SR Serialisierbare Historien

RC Rücksetzbare Historien

ACA Historien ohne kaskadierendes Rücksetzen

ST Strikte Historien

Abkürzungen

RC	ReCoverable
ACA	Avoid Cascading Abort
ST	STrict
SR	SeRializable

Anforderungen an die Concurrency Control eines DBMS

- Mindestanforderung: Die Einzeloperationen mehrerer Transaktionen sollten so gereiht werden, dass die resultierende Historie *serialisierbar* ist.
- Üblicherweise werden nur Historien in $ST \cap SR$ erzeugt.
- Realisierung:
  - am weitesten verbreitet: sperrbasierte Synchronisation
  - weitere Methoden: Zeitstempel-basierte Synchronisation, optimistische Synchronisation

Zwei-Phasen-Sperrprotokoll (2PL)

- Sperrbasierte Synchronisation
- Zwei-Phasen-Sperrprotokoll
- Striktes Zwei-Phasen-Sperrprotokoll

Sperrbasierte Synchronisation

Verträglichkeitsmatrix (= Kompatibilitätsmatrix):

	NL	S	X
S	✓	✓	–
X	✓	–	–

Zwei Sperrmodi:

- S Shared, read lock, Lesesperre
- X eXclusive, write lock, Schreibsperre

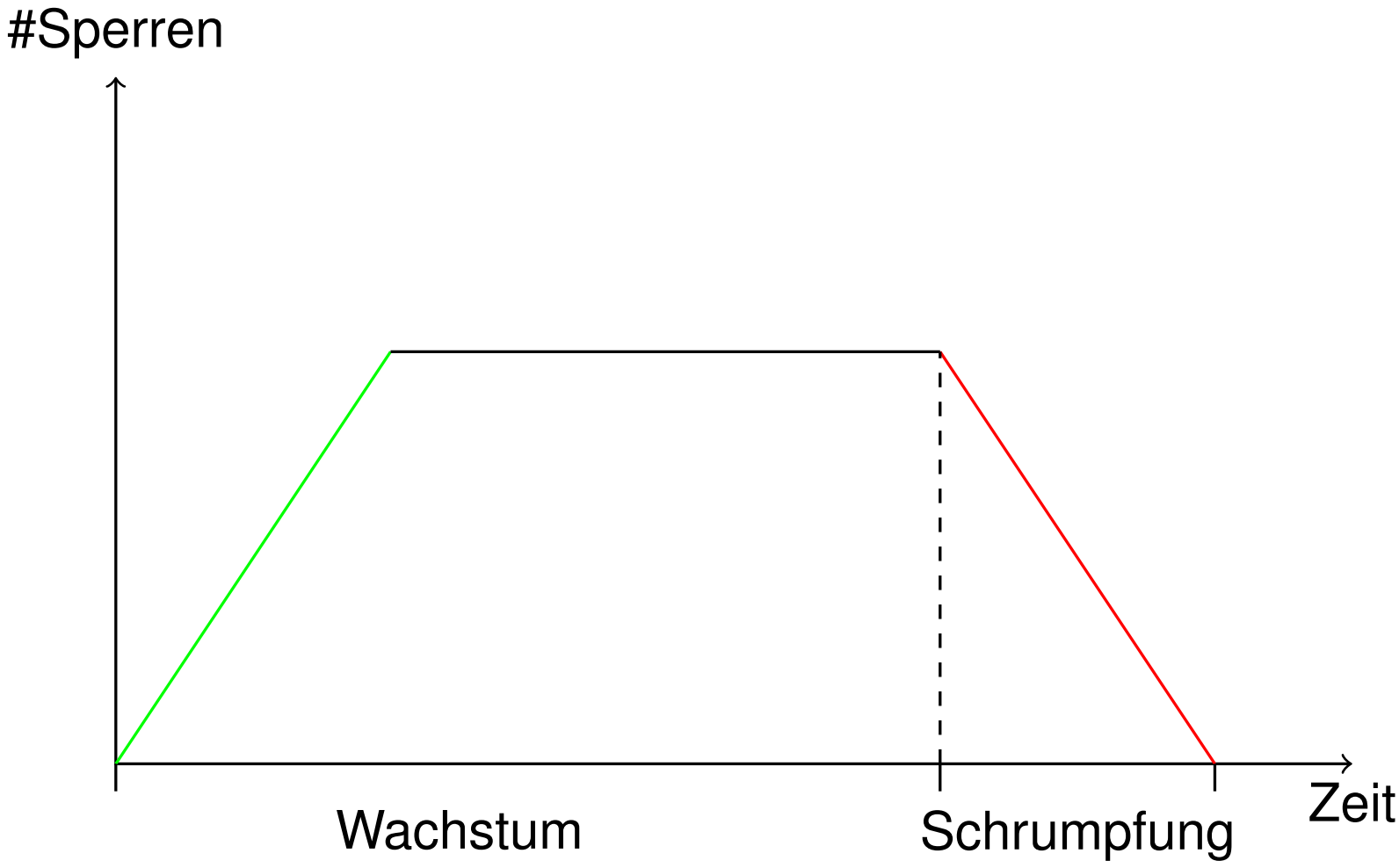
Weiterer möglicher Zustand:

- NL No Lock, Objekt im Moment nicht gesperrt

Zwei-Phasen-Sperrprotokoll

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 erhält eine Sperre auf ein benötigtes Objekt gemäß der Verträglichkeitstabelle. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine 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 *Schrumpfungsphase*, 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



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

Beispiel

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**	

Eigenschaften der 2PL Historien

2PL garantiert Serialisierbarkeit:

- Die Reihenfolge der TAs in einer äquivalenten seriellen Historie ergibt sich aus der Reihenfolge der Sperranforderungen bei Konflikten.
- Widersprüchliche Reihenfolgen dieser Sperranforderungen führen zu einem Deadlock.

2PL garantiert nicht Rücksetzbarkeit:

- Da die Sperren vor EOT freigegeben werden können, kann eine andere TA eventuell auf ein modifiziertes DB-Objekt zugreifen und früher committen (siehe Beispiel auf nächster Folie).

Nicht rücksetzbare Historie bei 2PL

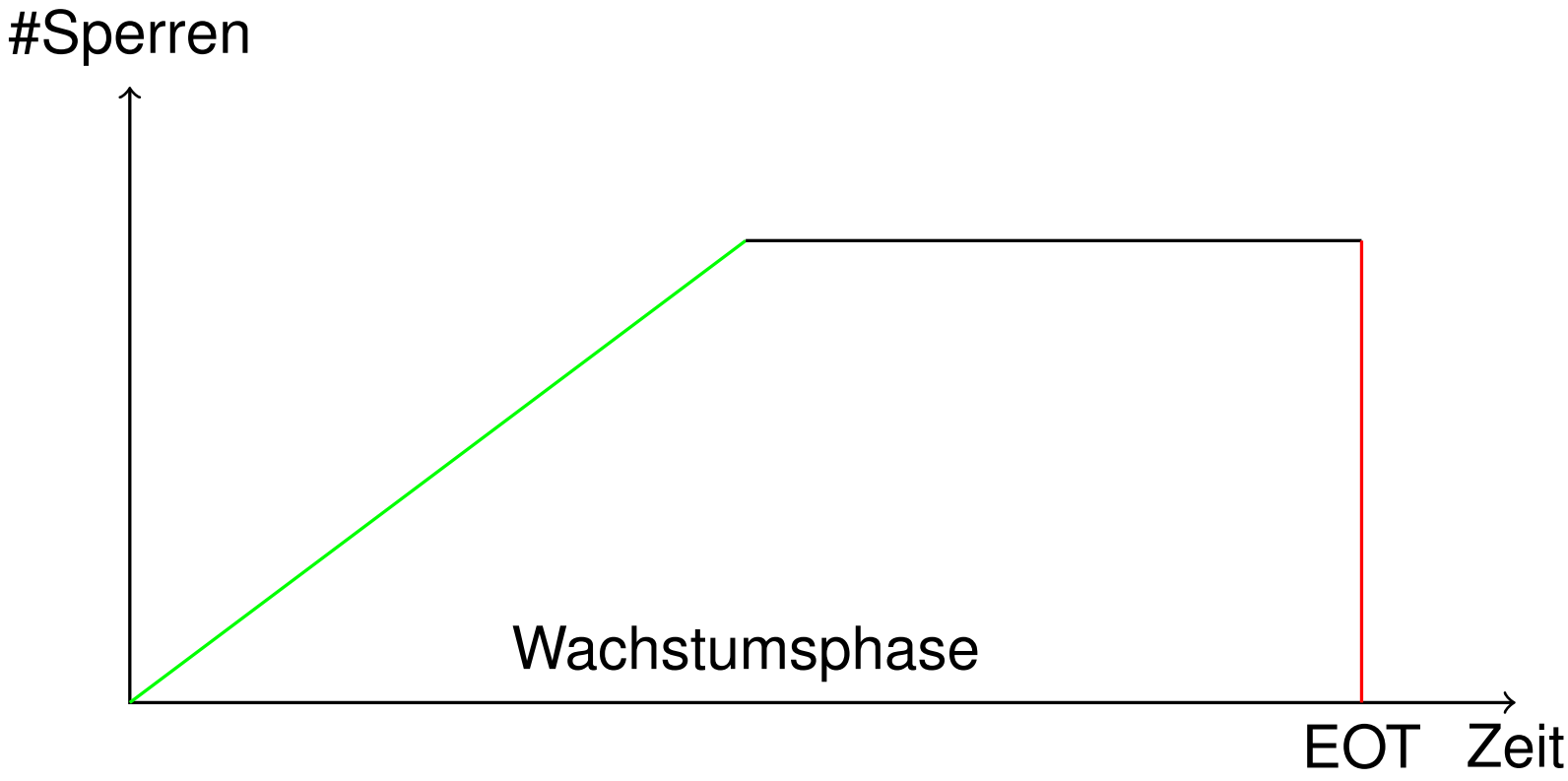
Beispiel

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockX**(A)	T_2 muss warten
7.	**lockX**(B)		
8.	**unlockX**(A)		T_2 wecken
9.	read(B)	read(A)	„T_2 liest von T_1“
10.		write(A)	
11.		**unlockX**(A)	
12.	write(B)	**commit**	
13.	**unlockX**(B)		
14.	**abort**		T_2 kann nicht mehr zurückgesetzt werden!

Strenges Zwei-Phasen-Sperrprotokoll

Erweiterung von 2PL zum strengen 2PL-Protokoll (strict 2PL):

- Alle Sperren werden bis EOT gehalten.
- Das strenge 2PL-Protokoll lässt nur strikte Historien zu.



Verklemmungen (Deadlocks)

- Deadlock-Erkennung
- Deadlock-Vermeidung

Verklemmungen (Deadlocks)

Beispiel

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*

Deadlock-Erkennung

Einfachste Methode: *Time-out Strategie*

- Idee: Abort einer Transaktion, wenn sie innerhalb einer vorgegeben Zeit keine Fortschritte macht.
- Problem: „gute“ Wahl des Timers, d.h.: falls Timer zu klein „Fehlalarme“, falls Timer zu groß können Deadlocks lange dauern.

Exakte Methode: *Wartegraph*

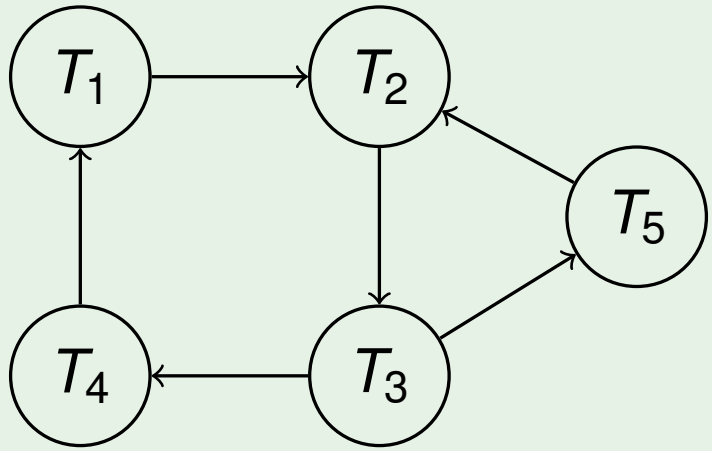
- Graph: Knoten sind aktive Transaktionen, Kante $T_i \rightarrow T_j$ falls T_i auf T_j wartet
- Deadlock genau dann, wenn der Wartegraph einen Zyklus enthält.

Deadlock-Erkennung

Beispiel

Wartegraph mit zwei Zyklen:

1. $T_1 \to T_2 \to T_3 \to T_4 \to T_1$
2. $T_2 \to T_3 \to T_5 \to T_2$



Beide Zyklen können durch Rücksetzen von T_3 aufgelöst werden.

Deadlock-Vermeidung

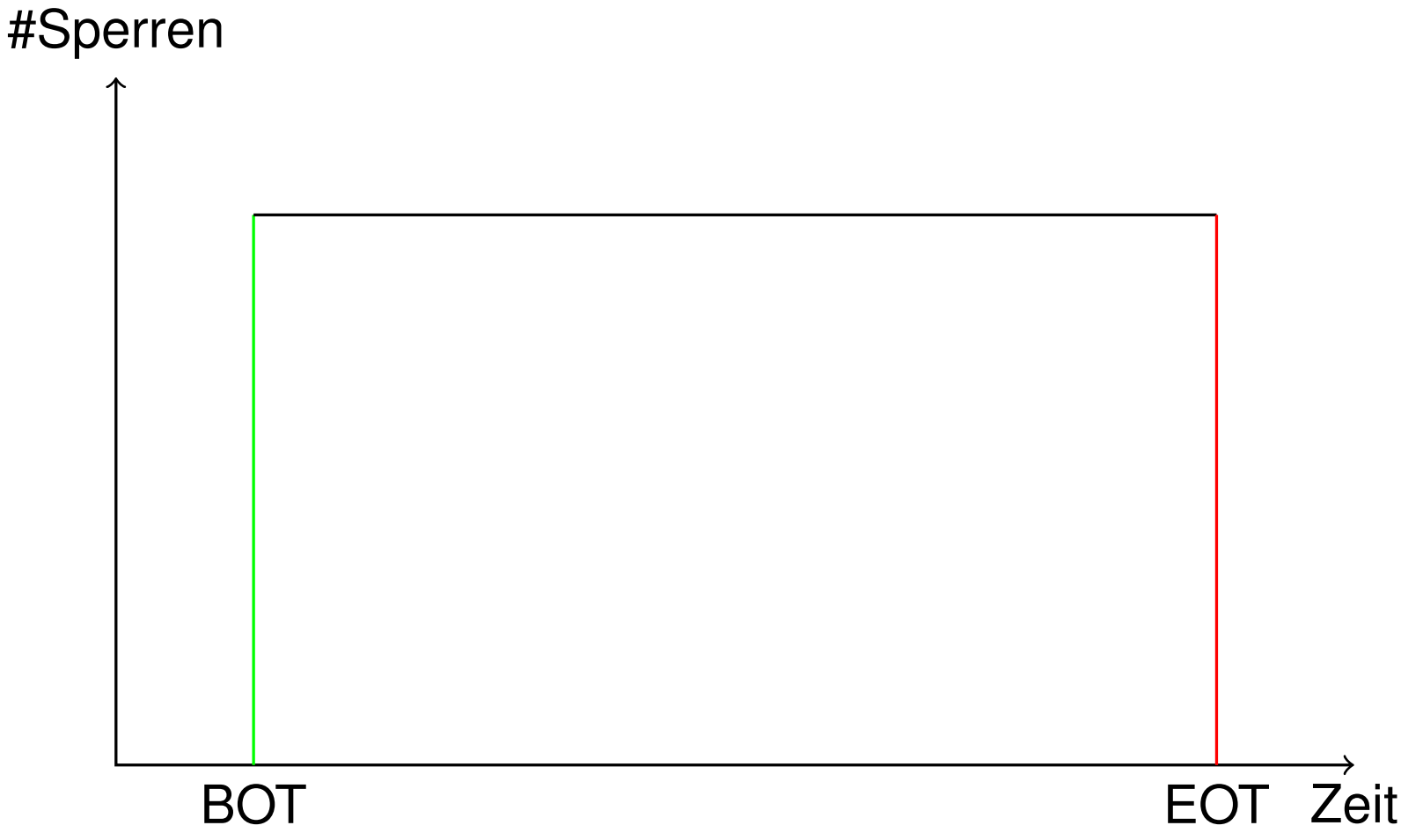
Deadlock-Vermeidung mittles „Preclaiming“:

- Idee: Eine Transaktion fordert bei BOT bereits alle Sperren an, die sie jemals brauchen wird, und hält die Sperren bis EOT.
- Variante von strict 2PL: „Conservative 2PL“
- Problem: Benötigte Sperren sind im allgemeinen zu Transaktionsbeginn nicht (exakt) bekannt.

Deadlock-Vermeidung mittles Zeitstempelverfahren:

- Jede Transaktion bekommt einen Zeitstempel
- Regel: Im Zweifelsfall wird eine jüngere TA abgebrochen

Conservative 2PL (Preclaiming)



Alle Sperren werden bei BOT angefordert und bis EOT gehalten.

Zwei Strategien bei Zeitstempelverfahren

T_1 will eine Sperre erwerben, die von T_2 gehalten wird:

- *wound-wait* Strategie
  - 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.
  - Idee: Eine ältere TA wartet niemals auf eine jüngere.
- *wait-die* Strategie
  - Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre.
  - Sonst wird T_1 abgebrochen und zurückgesetzt.
  - Idee: Eine jüngere TA wartet niemals auf eine ältere.

Multiple-Granularity Locking (MGL)

- Idee: unterschiedliche Sperrgranulate
- Erweiterte Sperrmodi
- Sperrprotokoll

MGL: Multiple-Granularity Locking

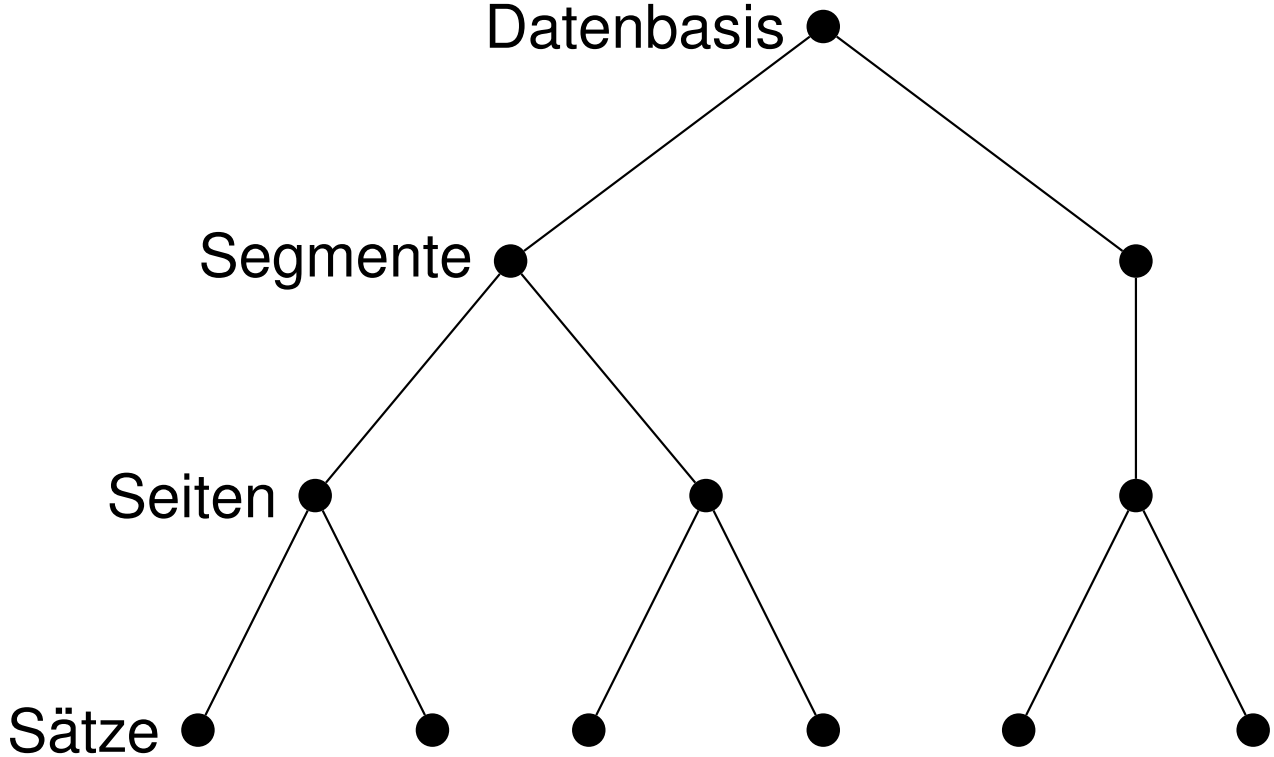
Idee: Einheitliche Sperrgranulate (z.B.: Sperre pro Datensatz, Sperre pro Seite, etc.) können ineffizient sein:

- zu kleine Granularität: ineffizient bei Transaktion mit vielen Datenzugriffen.
- zu groß: blockiert eventuell mehr Transaktionen als nötig.

Mehr Flexibilität durch unterschiedliche Sperrgranulate.
Üblicherweise:

- DBMS wählt passende Granularität (Satz, Seite, Tabelle)
- „Lock escalation“: Transaktion sperrt anfangs mit kleiner Granularität und — wenn immer mehr Sperren benötigt werden — fordert Sperren mit höherer Granularität an.

Hierarchische Anordnung möglicher Sperrgranulate



Segment: mehrere (logisch zusammengehörende) Seiten (insbesondere Tabelle)

Erweiterte Sperrmodi

- Problem: Ohne weitere Vorkehrungen müssten immer alle darunter liegenden Objekte überprüft werden, bevor eine Sperre auf einer höheren Ebene gewährt werden kann.
- Lösung: Zusätzliche Sperrmodi (IS und IX), die anzeigen, dass in der Hierarchie weiter unten eine bestimmte Sperre existiert.
- Sperrmodi bei MGL:

NL	keine Sperre (No Lock)
S	Lesesperre (Shared lock)
X	Schreibsperre (eXclusive lock)
IS	Intention Shared lock: weiter unten in der Hierarchie ist eine Lesesperre (S) beabsichtigt
IX	Intention eXclusive lock: weiter unten in der Hierarchie ist eine Schreibsperre (X) beabsichtigt

Kompatibilitätsmatrix

	NL	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

Sperrprotokoll des MGL

„top-down“ Anforderung der benötigten Sperren:

- Bevor eine TA einen Knoten mit S oder IS sperren kann, benötigt diese TA für alle Vorgänger in der Hierarchie eine IS- oder IX-Sperre.
- Bevor eine TA einen Knoten mit X oder IX sperren kann, benötigt diese TA für alle Vorgänger eine IX-Sperre.

„bottom-up“ Freigabe der Sperren:

- Die IS- bzw. IX-Sperre an einem Knoten darf erst freigegeben werden, wenn alle Sperren auf Nachfolgerknoten bereits freigegeben worden sind.

Datenbasishierarchie mit Sperren

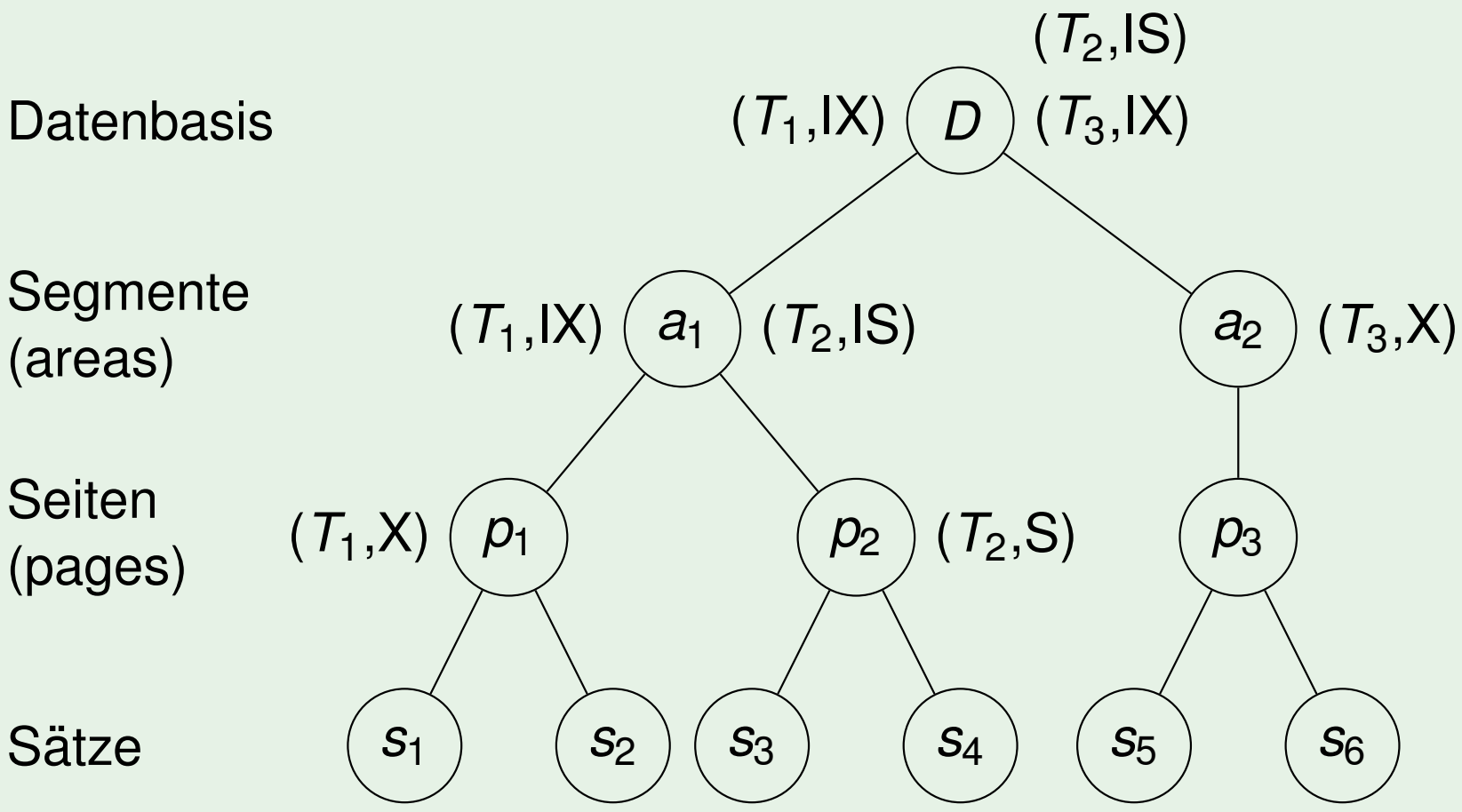
Beispiel

3 Transaktionen:

- T_1 exclusive lock für Seite p_1 unterhalb von Segment a_1
- T_2 shared lock für Seite p_2 unterhalb von Segment a_1
- T_3 exclusive lock für Segment a_2

Datenbasishierarchie mit Sperren

Beispiel



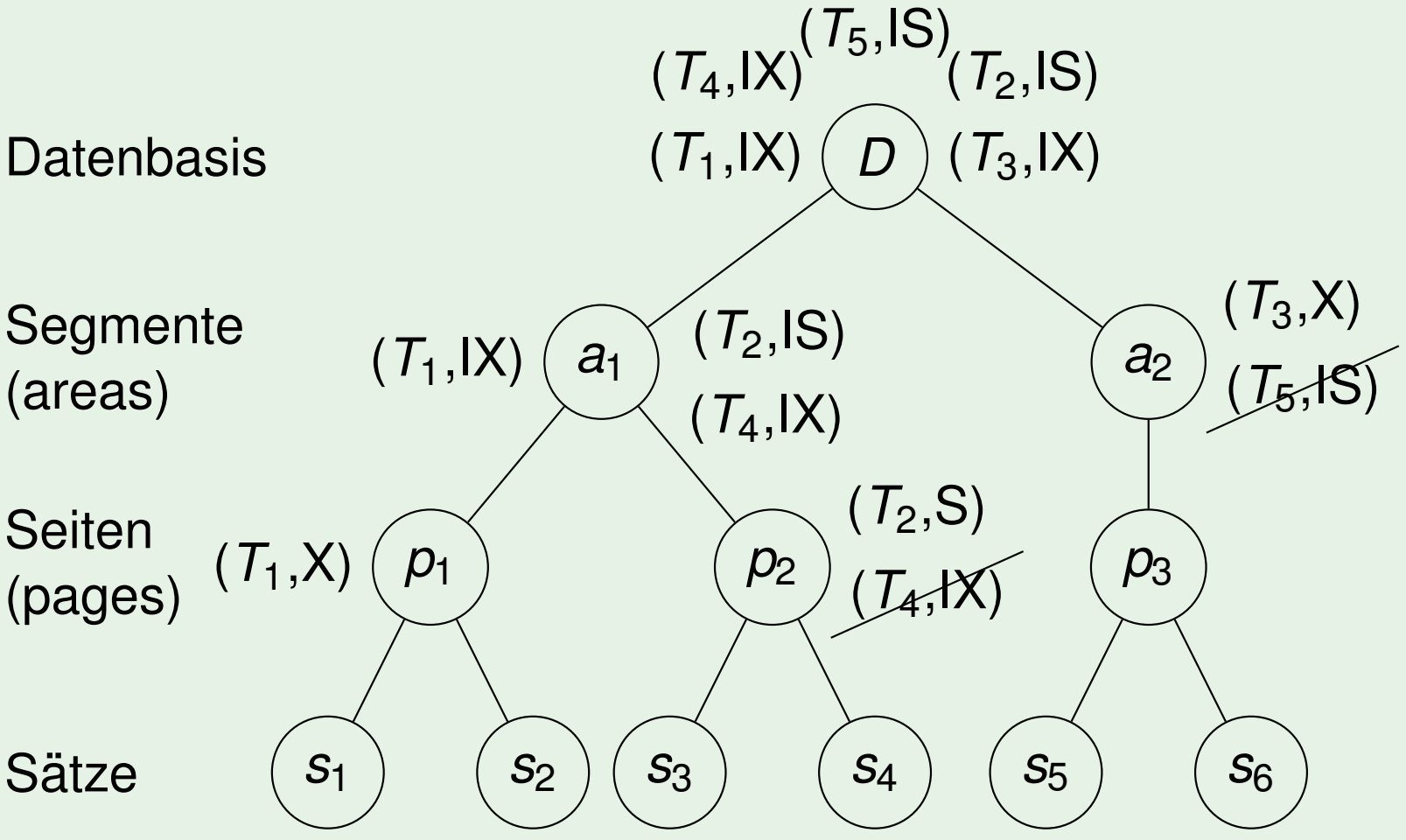
Datenbasishierarchie mit blockierten Transaktionen

Beispiel

- 3 Transaktionen:
  - T_1 exclusive lock für Seite p_1 unterhalb von Segment a_1
  - T_2 shared lock für Seite p_2 unterhalb von Segment a_1
  - T_3 exclusive lock für Segment a_2
- Fortsetzung:
  - T_4 exclusive lock für Satz s_3 unterhalb von Seite p_2: IX-lock Anforderung für p_2 scheitert wegen S-Sperre von T_2.
  - T_5 shared lock für Satz s_5 unterhalb von Seite p_3 unterhalb von Segment a_2: IS-lock Forderung für a_2 scheitert wegen X-Sperre von T_3.

Datenbasishierarchie mit blockierten Transaktionen

Beispiel



Insert/Delete-Operationen

- Sperren beim Einfügen / Löschen
- Phantomproblem

Sperren beim Einfügen / Löschen

Löschen:

- Für das Löschen eines Objekts benötigt eine Transaktion eine X-Sperre. Diese X-Sperre wird bis EOT gehalten.
- Eine andere TA, die für dieses Objekt ebenfalls eine Sperre erwerben will, wird diese nicht mehr erhalten, falls die Löschtransaktion erfolgreich (mit commit) abschließt.

Einfügen:

- Für das Einfügen eines Objekts benötigt eine Transaktion eine X-Sperre. Diese X-Sperre wird bis EOT gehalten.
- Das Phantomproblem erfordert zusätzliche Maßnahmen.

Phantomproblem

Beispiel

T_1	T_2
SELECT COUNT(∗)	
FROM pruefen	
WHERE Note **BETWEEN** 1 **AND** 2	
	INSERT INTO pruefen
	VALUES(29555,5001,2137,1)
SELECT COUNT(∗)	
FROM pruefen	
WHERE Note **BETWEEN** 1 **AND** 2	

Phantomproblem

Problem:

- Auch beim strengen 2PL Protokoll erwirbt eine Transaktion nur Sperren auf Datensätze, die zu einem bestimmten Zeitpunkt in der DB existieren.
- Damit wird aber nicht verhindert, dass „Phantome“ von anderen Transaktionen eingefügt werden.

Lösung: Das Anlegen neuer Datensätze muss verhindert werden.

- Tabelle ohne Index: Anlegen neuer Seiten/Datensätze verbieten (z.B. IS-Sperre auf Tabelle)
- Tabelle mit Index: Indexbereichssperren zusätzlich zu den Tupelsperren.

Weitere Synchronisationsmethoden

- Zeitstempel-basierende Synchronisation
- Optimistische Synchronisation
- Synchronisation von Indexstrukturen

Zeitstempel-basierende Synchronisation

Idee:

- Jeder Transaktion T_i wird bei BOT ein Zeitstempel zugewiesen: $TS(T_i)$.
- Jedem Datum A in der Datenbasis werden zwei Marken zugeordnet: $readTS(A)$, und $writeTS(A)$.
- Wenn eine TA T_i auf ein Datum A zugreifen will, wird $TS(T_i)$ mit $readTS(A)$ bzw. $writeTS(A)$ verglichen.
- Bei einem Konflikt wird eine TA rückgesetzt.

Ergebnis:

- Serialisierbare, Deadlock-freie Historien.
- Reihenfolge der TAs in serieller Historie entspricht den Zeitstempeln.

Zeitstempel-basierende Synchronisation

Lesezugriff

T_i will A lesen, also $r_i(A)$:

- Fall 1: $TS(T_i) < writeTS(A)$:
  - d.h.: Die Transaktion T_i ist älter als eine andere TA, die auf A schon geschrieben hat.
  - In diesem Fall wird T_i zurückgesetzt.
- Fall 2: $TS(T_i) \geq writeTS(A)$:
  - d.h.: Die Transaktion T_i ist jünger als die letzte TA, die auf A geschrieben hat.
  - T_i kann die Leseoperation durchführen.
  - Die Marke $readTS(A)$ wird auf $\max(TS(T_i), readTS(A))$ gesetzt.

Zeitstempel-basierende Synchronisation

Schreibzugriff

T_i will A schreiben, also $w_i(A)$:

- Fall 1: $TS(T_i) < readTS(A)$:
  - d.h.: Es gab eine jüngere TA, die den neuen Wert von A, den T_i gerade schreiben möchte, hätte lesen müssen.
  - In diesem Fall wird T_i zurückgesetzt.
- Fall 2: $TS(T_i) < writeTS(A)$:
  - d.h.: Es gab eine jüngere TA, die auf A geschrieben hat und die nun von T_i überschrieben würde.
  - In diesem Fall wird T_i zurückgesetzt.
- Sonst: T_i darf auf A schreiben und die Marke $writeTS(A)$ wird auf $TS(T_i)$ gesetzt.

Optimistische Synchronisation

3 Phasen:

1. Lesephase:
   - Alle Operationen der Transaktion werden ausgeführt.
   - Schreiboperationen nur auf lokalen Kopien der Daten
2. Validierungsphase:
   - Prüfung, ob die TA committed werden darf.
   - Mögliche Konflikte werden mittels Zeitstempel (bei BOT, Validierung und Schreiben) erkannt.
3. Schreibphase:
   - Die Änderungen dieser TA werden in die DB eingebracht.

Optimistische Synchronisation

Idee:

- Alle Operationen aller TAs werden vollständig ausgeführt. Schreiboperationen erfolgen aber auf lokalen Variablen.
- Erst am Ende wird geprüft, ob ein Konflikt vorliegt. In diesem Fall wird die TA zurückgerollt. Ansonsten werden die Änderungen in die DB übernommen.
- Validierungs- und Schreibphase dürfen nicht unterbrochen werden, d.h. immer nur eine TA in diesen Phasen.

Ergebnis:

- Serialisierbare, Deadlock-freie Historien.
- Reihenfolge der TAs in serieller Historie entspricht den Zeitstempeln der Validierungsphase.

Validierung bei der optimistischen Synchronisation

Validierung der Transaktion T_j: für alle TAs T_a, die vor T_j die Validierung abgeschlossen haben, muss eine der folgenden beiden Bedingungen gelten:

1. T_a war zum Beginn der Transaktion T_j schon abgeschlossen (und zwar einschließlich der Schreibphase).
2. $\mathit{WriteSet}(T_a) \cap \mathit{ReadSet}(T_j) = \emptyset$
   - $\mathit{WriteSet}(T_a) =$ alle von T_a geschriebenen Datenelemente
   - $\mathit{ReadSet}(T_j) =$ alle von T_j gelesenen Datenelemente

Bemerkung: W-W Konflikte kann es keine geben, da die TAs vor der Schreibphase nur auf lokalen Variablen schreiben.

Synchronisation von Indexstrukturen (B^+-Bäume)

Grundsätzlich möglich:

- Behandlung von Indexstrukturen wie „normale“ Daten.
- Nachteil: Auf den oberen Ebenen eines B^+-Baums würden häufig Konflikte auftreten.

Lockerung des strict 2PL-Verfahrens bei B^+-Bäumen, weil:

- Die oberen Ebenen dienen nur der Navigation; die eigentlichen Daten (bzw. TIDs) sind nur an den Blättern. D.h., dauerhafte Sperren letztlich nur an den Blättern.
- Bei Einfügeoperationen ist eine Schreibsperre an inneren Knoten nur dann erforderlich, wenn sich ein Überlauf bis zu diesem Knoten fortsetzt.

Synchronisation von Indexstrukturen (B^+-Bäume)

„Lock coupling“ (= Anforderung einer neuen Sperre und Freigabe der alten Sperre):

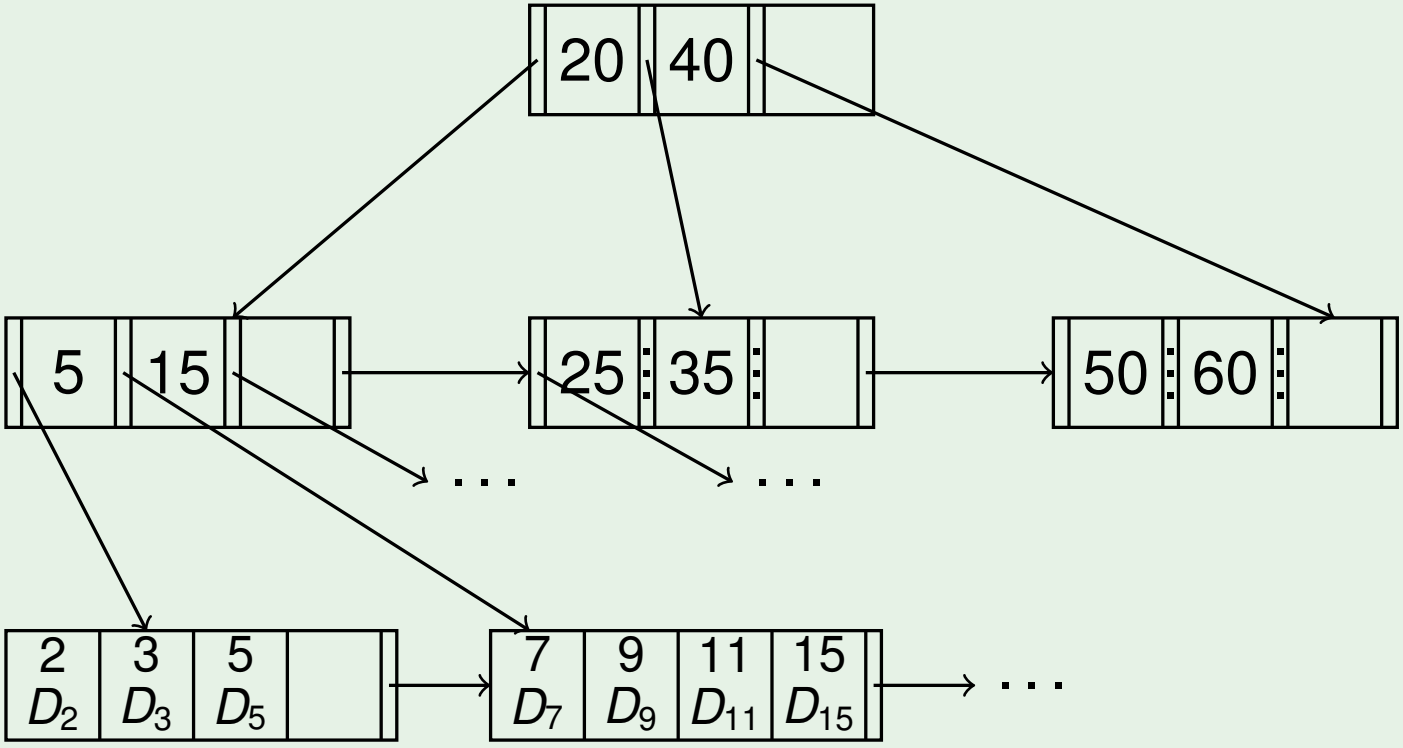
- Beim Abstieg im Baum (Suchen, Einfügen, Löschen) wird für jeden Knoten kurz ein shared lock angefordert, das beim Übergang zum nächsten Knoten wieder freigegeben wird.

Erweiterung der B^+-Bäume um „rechts“-Verweise:

- Auf jeder Stufe des Baums sind Geschwisterknoten mittels „rechts“-Verweisen miteinander verknüpft.
- Beim Navigieren einer TA T_1 kann ein Knoten wegen einer Einfügeoperation einer anderen TA T_2 aufgespaltet werden. D.h. Suche von T_1 muss eventuell beim rechten Nachbarn (und nicht beim Kindknoten) fortgesetzt werden.

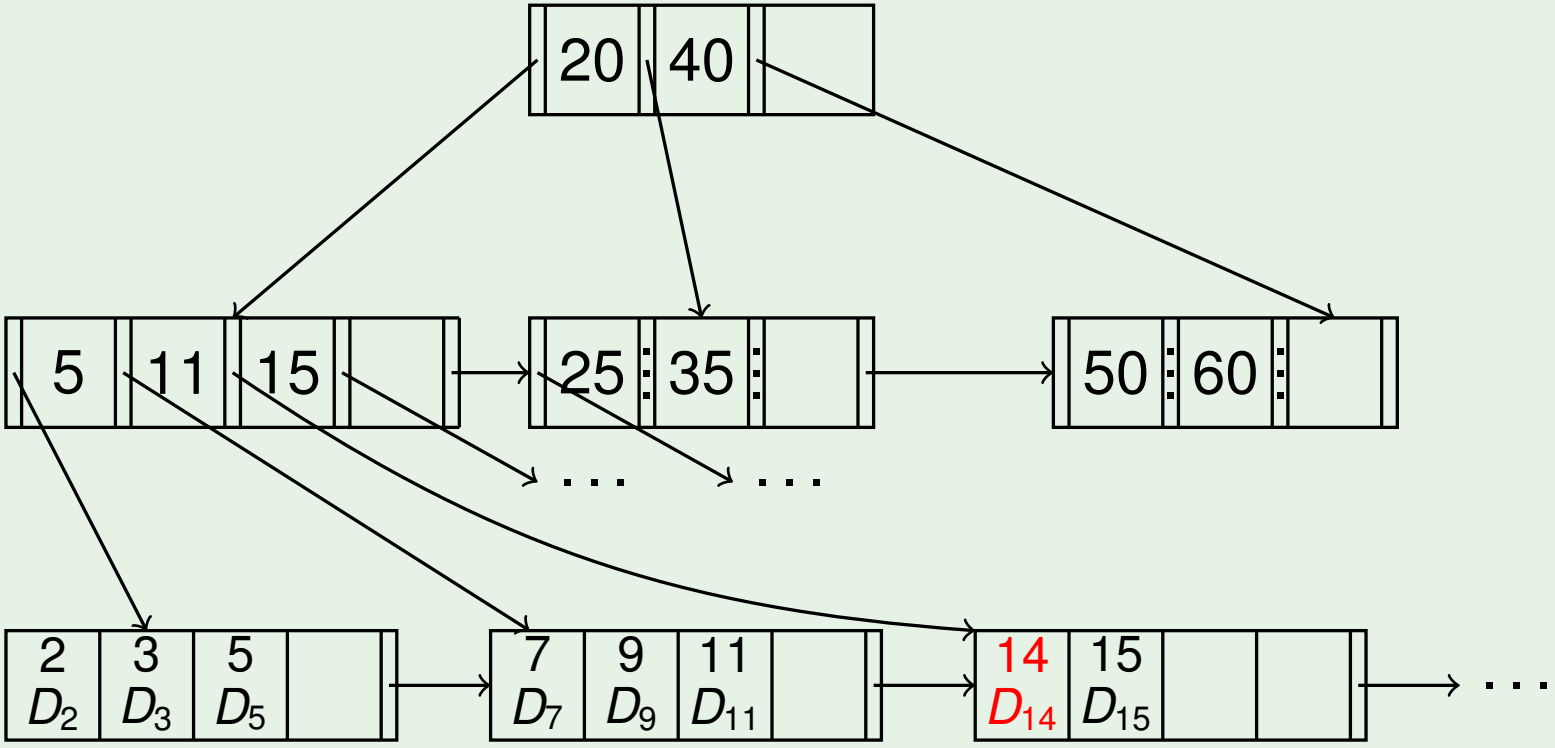
B^+-Baum mit *rechts*-Verweisen

Beispiel



B^+-Baum mit *rechts*-Verweisen

Beispiel



Beispiel

2 Transaktionen T_1 und T_2:

- T_1 sucht den Eintrag mit Schlüsselwert 15.
- T_2 fügt den Wert 14 ein.

Dynamischer Ablauf:

- T_1 ist bereits bis zum inneren Knoten mit den Werten 5 und 15 navigiert.
- T_2 fügt den Wert 14 ein, d.h. Aufspalten des Blattknotens.
- T_1 navigiert weiter zum ursprünglich ermittelten Blattknoten (2. Blattknoten von links).
- Da der Wert 15 nicht mehr hier ist, navigiert T_1 weiter zum rechten Geschwisterknoten.

Transaktionsverwaltung in SQL

- **SET TRANSACTION** Kommando

SET TRANSACTION Kommando

Access Mode:

- read only: TA hat nur Lesezugriffe (und benötigt daher nur Lesesperren)
- read write (= default)

Isolation Level: Unterschiedlich starker Schutz vor Anomalien, um Parallelisierbarkeit zu erhöhen

- Niedrigste Stufe (nur bei read only): read uncommitted nur sinnvoll, um einen allgemeinen Überblick über die DB zu bekommen
- Höchste Stufe (= default): serializable garantiert Serialisierbarkeit

SET TRANSACTION Kommando

```
SET TRANSACTION
   [read only, | read write,]
   [isolation level
      read uncommitted, |
      read committed,   |
      repeatable read,  |
      serializable,]
   [diagnostics size ...,]
```

Isolation Levels und mögliche Anomalien

Isolation Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	möglich	möglich	möglich
READ COMMITTED	–	möglich	möglich
REPEATABLE READ	–	–	möglich
SERIALIZABLE	–	–	–

Zusammenfassung

Protokolle und ihre wichtigsten Eigenschaften

Protokoll	Äquivalente serielle Historie	Weitere Eigenschaften	Deadlock möglich
2PL	Reihenfolge der Sperranforderungen bei Konflikten	im allgemeinen nicht rücksetzbar	ja
Strict 2PL	wie 2PL	strikt	ja
Strict 2PL + Deadlock-Vermeidung	wie 2PL	strikt	nein
Zeitstempel-basierend	Zeitpunkt von BOT	im allgemeinen nicht rücksetzbar	nein
optimistisch	Zeitpunkt der Validierung	strikt	nein