

# Architektur und Implementierung von Datenbanksystemen WS 05/06

Dr. Jens-Peter Dittrich

[jens.dittrich@inf](mailto:jens.dittrich@inf)

[www.inf.ethz.ch/~jensdi](http://www.inf.ethz.ch/~jensdi)

Institut für Informationssysteme



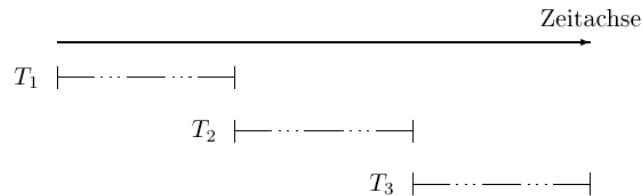
# Mehrbenutzersynchronisation



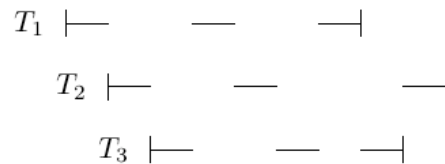
# Mehrbenutzersynchronisation

(ACID: Wie wird Isolation realisiert?)

- Ausführen der drei Transaktionen  $T_1$ ,  $T_2$ ,  $T_3$ 
  - im Einbenutzerbetrieb



- im Mehrbenutzerbetrieb



# Verlorengegangene Änderungen (lost updates)

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( $B, b_1$ )	

- Effekt von  $T_2$  geht verloren

## 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.	<b>abort</b>	

- wird auch als “dirty read” bezeichnet
- Transaktion  $T_2$  wurde auf Basis inkonsistenter Daten durchgeführt

## Phantomproblem

$T_1$	$T_2$
	select sum(KontoStand) from Konten
insert into Konten values ( $C, 1000, \dots$ )	
	select sum(KontoStand) from Konten

- $T_2$  führt dasselbe SQL statement zweimal aus mit unterschiedlichem Ergebnis
- der neueingefügte Wert von  $T_1$  wird als “Phantom” bezeichnet

## Serielle Ausführung von $T_1$ und $T_2$ : $T_1|T_2$

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>

## Serialisierbarkeit

- Serialisierbare Historie von  $T_1$  und  $T_2$ 
  - Historie ist äquivalent zu einer seriellen Historie  $T_1|T_2$
  - dennoch parallele (verzahnte) Ausführung möglich
  - Datenbasis ist in demselben Zustand wie bei einer seriellen Historie

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>

## Nicht serialisierbare Historie

- Bezogen auf das Datenobjekt A kommt  $T_1$  vor  $T_3$
- Bezogen auf das Datenobjekt B kommt  $T_3$  vor  $T_1$
- Diese Historie ist **nicht** äquivalent zu einer der beiden seriellen Ausführungen  $T_1|T_3$  oder  $T_3|T_1$

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 Überweisungstransaktionen

- Achtung: diese Historie ist **nicht** serialisierbar
- Trotzdem wäre die Datenbasis in diesem speziellen Fall konsistent!
- Das ist aber Zufall und liegt an der in diesem Fall verwendeten Anwendungssemantik!

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>	

## Zwei verzahnte Überweisungstransaktionen

- Achtung: diese Historie ist **nicht** serialisierbar
- In diesem Fall wäre die Datenbasis inkonsistent!
- Hier sorgt die Anwendungssemantik (\*) für eine inkonsistente Datenbasis!
- Aus Sicht des DBMS ist diese Semantik aber nicht erkennbar und darf nicht berücksichtigt werden!

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>	

## Rücksetzbare Historien (recoverable schedules)

- Zusätzliche Anforderung (neben Serialisierbarkeit): Jede Transaktion muss vor Ausführung eines commit lokal zurückgesetzt werden können.
- Eine Historie heisst rücksetzbar, falls immer die schreibende Transaktion  $T_{j \neq i}$  vor der lesenden Transaktion  $T_i$  ihr commit durchführt.
- Anders ausgedrückt: eine Transaktion  $T_i$  darf erst dann ihr commit durchführen, wenn alle Transaktionen  $T_{j \neq i}$ , von denen sie **gelesen** hat, beendet sind.

## Kaskadierendes Rücksetzen

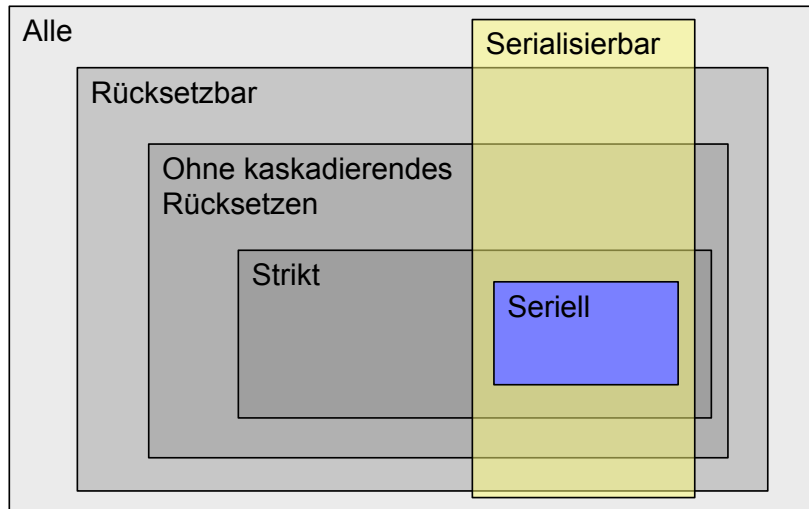
Schritt	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
0.	...				
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_4(D)$	
8.					$r_5(D)$
9.	$a_1$ (abort)				

- $T_2$  ist abhängig von  $T_1$ ,  $T_3$  ist abhängig von  $T_2$ , usw.
- In Schritt 9 bricht  $T_1$  ab.
- Folge: es müssen alle Transaktionen  $T_2$  bis  $T_5$  zurückgesetzt werden! (Schneeballeffekt)
- Abhilfe: Änderungen erst nach dem commit **lesen**.

## Strikte Historien (strict schedules)

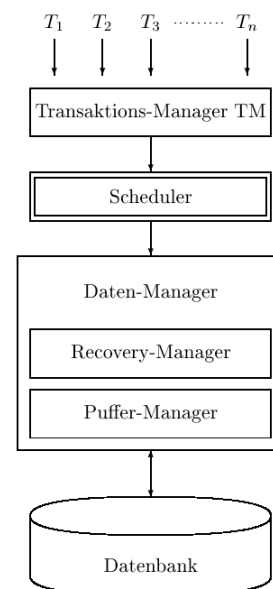
- Zusätzliche Anforderung (neben Vermeidung von Kaskadierendem Rücksetzen):  
veränderte Daten einer laufenden Transaktion dürfen nicht **überschrieben** werden
- D.h. ändert eine Transaktion  $T_1$  einen Wert A, darf dieser Wert von keiner anderen Transaktion gelesen oder überschrieben werden bis  $T_1$  abgeschlossen wurde durch commit
- siehe striktes Zwei-Phasen-Sperrprotokoll, exklusives Sperren

## Beziehungen zwischen den Historien

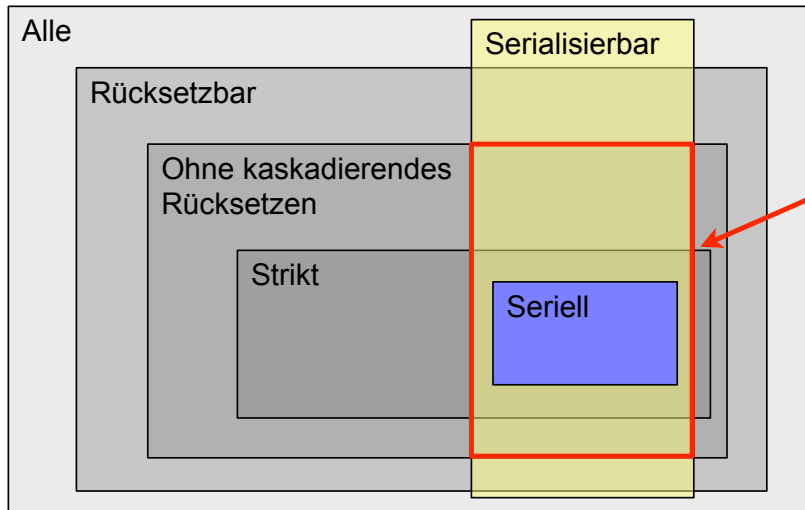


## Der Datenbank-Scheduler

- Muss Einzeloperationen so ausführen, dass die resultierende Historie serialisierbar ist
- Meist wird zusätzlich verlangt: Historie ohne kaskadierendes Rücksetzen
- im folgenden: Techniken für die Realisierung eines DB-Schedulers



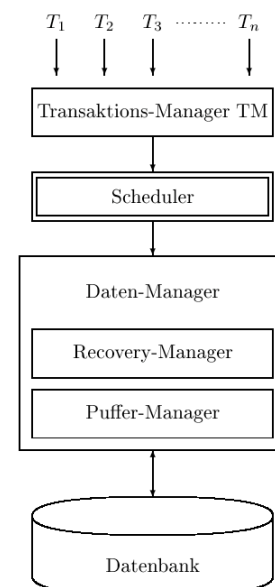
## Beziehungen zwischen den Historien



Scheduler verlangt diese Historien  
D.h. der Schnitt  
von  
serialisierbaren  
Historien und  
denjenigen ohne  
kaskadierendes  
Rücksetzen

## Implementierung des Datenbank-Scheduler

- Pessimistische Verfahren
  - sperrbasiert
  - zeitstempelbasiert
- Optimistische Verfahren
  - optimistische Synchronisation



## Sperrbasierte Synchronisation

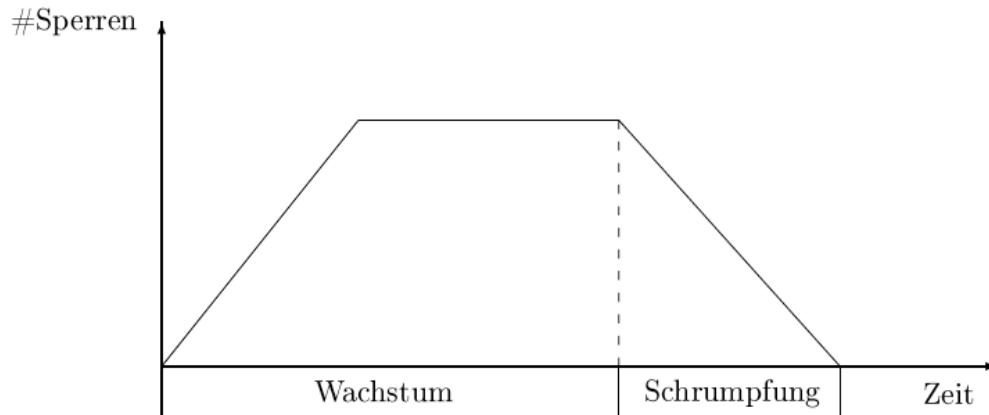
- Zwei Sperrmodi
  - S (**S**hared lock, read, Lesesperre)
  - X (**eX**clusive lock, write, Schreibsperre)
- Verträglichkeitsmatrix (Kompatibilitätsmatrix)

	<i>NL</i>	<i>S</i>	<i>X</i>
<i>S</i>	✓	✓	–
<i>X</i>	✓	–	–

## Zwei-Phasen-Sperrprotokoll: Definition

- Jedes Objekt, das von einer TA benutzt werden soll, muss vorher entsprechend gesperrt werden
- Eine TA fordert eine Sperre, die sie bereits besitzt, nicht erneut an.
- Eine TA muss andere Sperren gemäss Verträglichkeitsmatrix beachten. Kann die Sperre nicht gewährt werden, wird die A in eine Warteschlange eingereiht.
- Jede TA durchläuft zwei Phasen
  - Eine **Wachstumsphase**, in der sie Sperren anfordern kann aber keine freigeben darf
  - Eine **Schrumpfungphase**, in der sie ihre bisher erworbenen Sperren freigibt aber keine neuen anfordern darf
  - Bei EOT muss die TA alle ihre erworbenen Sperren zurückgeben

## 2PL



- 2PL garantiert Serialisierbarkeit

## 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)

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>	

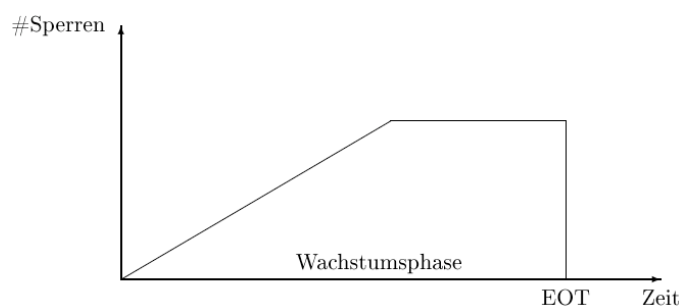
## Kaskadierendes Rücksetzen

- Was passiert, wenn  $T_1$  direkt vor Schritt 15 scheitert?
- Dann muss  $T_2$  zurückgesetzt werden, da  $T_2$  'dreckige' Daten von  $T_1$  gelesen hat!
- 2PL vermeidet dies also nicht!

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

- Problem: 2PL schliesst kaskadierendes Rücksetzen nicht aus
- Erweiterungen zum **strengen** 2PL
  - alle Sperren werden bis EOT gehalten
  - es gibt keine Schrumpfungsphase mehr
  - damit ist kaskadierendes Rücksetzen (Schneeballeffekt) ausgeschlossen



## Verklemmungen (Deadlocks)

- Ein verklemmter Schedule

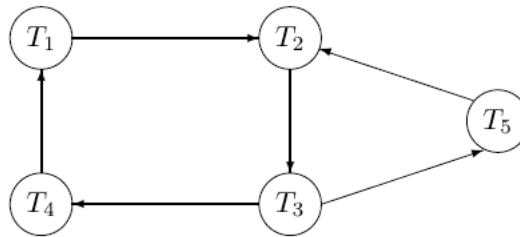
Schritt	$T_1$	$T_2$	Bemerkung
1.	<b>BOT</b>		
2.	<b>lockX(A)</b>		
3.		<b>BOT</b>	
4.		<b>lockS(B)</b>	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	<b>lockX(B)</b>		$T_1$ muß warten auf $T_2$
9.		<b>lockS(A)</b>	$T_2$ muß warten auf $T_1$
10.	...	...	$\Rightarrow$ <i>Deadlock</i>

## Erkennung von Verklemmungen: Timeout

- Timeout-Verfahren
  - Falls die TA nach einem gewissen Zeitintervall keinen Fortschritt erzielt hat wird die TA abgebrochen
- Problem: Wahl des Zeitintervalls
  - zu klein: zu viele TAs unnötig abgebrochen
  - zu gross: Sperren werden zulange geduldet

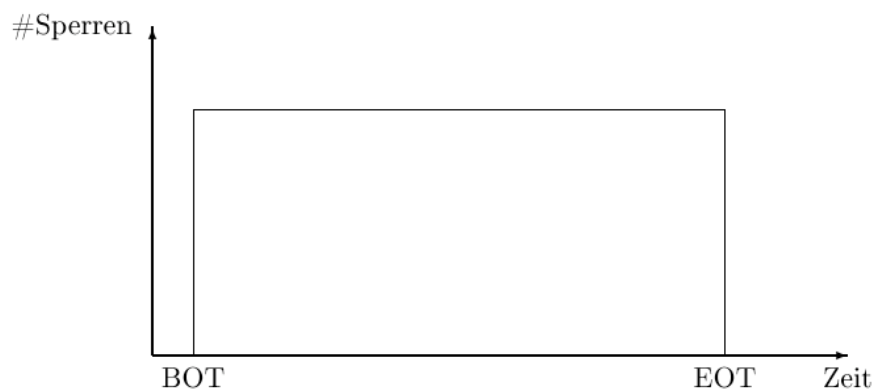
## Erkennung von Verklemmungen: Wartegraph

- 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

## Vermeidung von Verklemmungen: Preclaiming



- Wie weiss die TA vorher, welche Datenobjekte sie benötigt?
- Deswegen in der Praxis nicht realisierbar
- (Analogie: Preplanning im DB-Puffer, Kap. 2, Folie 38)

## Verklemmungsvermeidung durch Zeitstempel

- Jeder TA wird ein eindeutiger Zeitstempel (TS) zugeordnet
- TAs dürfen nicht mehr “bedingungslos” auf eine Sperre warten
- **wound-wait** Strategie

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

Falls  $TS(T_1) < TS(T_2)$ : //“ $T_1$  älter  $T_2$ ?”

$T_2$  wird abgebrochen und zurückgesetzt

$T_1$  erhält die Sperre

**wound**

Sonst

$T_1$  wartet auf die Freigabe der Sperre durch  $T_2$

**wait**

bevorzuge  
kleine TS,  
d.h. ältere  
TAs

- **wait-die** Strategie

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

Falls  $TS(T_1) < TS(T_2)$ : //“ $T_1$  älter  $T_2$ ?”

$T_1$  wartet auf die Freigabe der Sperre durch  $T_2$

**wait**

Sonst

$T_1$  wird abgebrochen und zurückgesetzt

$T_2$  erhält die Sperre

**die**

bevorzuge  
grosse TS  
d.h. jüngere  
TAs

## Verklemmungsvermeidung durch Zeitstempel

- Zeitstempelmethode ist garantiert verklemmungsfrei
- Nachteile
  - zuviele TAs werden unnötig zurückgesetzt
  - wound-wait: junge TAs haben das Nachsehen
  - wait-die: alte TAs haben das Nachsehen

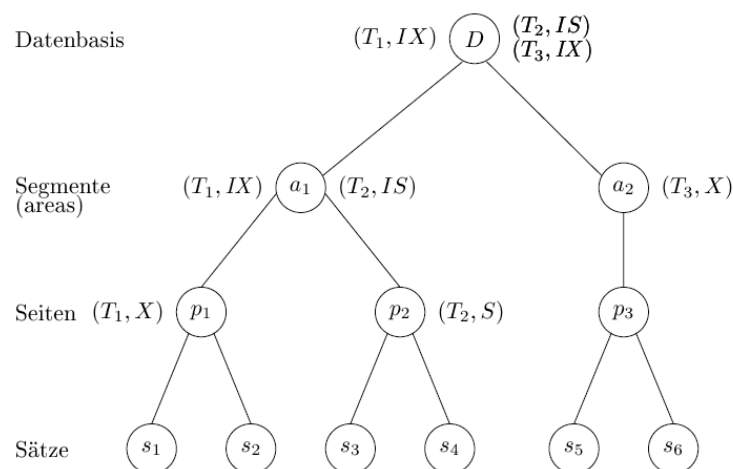


## MGL: Sperrprotokoll

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

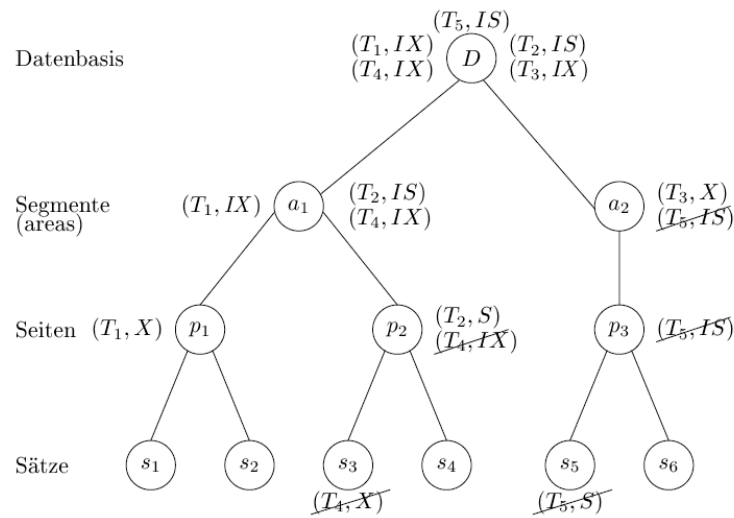
	NL	S	X	IS	IX
S	✓	✓	-	✓	-
X	✓	-	-	-	-
IS	✓	✓	-	✓	✓
IX	✓	-	-	✓	✓

## Datenbasis-Hierarchie mit Sperren



- T<sub>1</sub> will Seite p<sub>1</sub> exklusiv sperren
- T<sub>2</sub> will Seite p<sub>2</sub> mit S-Sperre belegen
- T<sub>3</sub> will Segment a<sub>2</sub> mit X sperren

## Datenbasis-Hierarchie mit Sperren



- T<sub>4</sub> und T<sub>5</sub> sind blockiert (warten auf Freigabe von Sperren)
- T<sub>4</sub> und T<sub>5</sub> sind blockiert aber nicht verklemmt
- Achtung: auch bei MGL können Verklemmungen auftreten!

## Einfüge- und Löschoperationen, Phantome

- Naheliegende Methode
  - Vor dem Löschen eines Objektes muss die TA 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 Objektes erwirbt die einfügende TA eine X-Sperre.
- Funktionert aber leider nicht

## Phantomprobleme

$T_1$	$T_2$
<pre>select count(*) from prüfen where Note between 1 and 2;</pre>	
<pre>select count(*) from prüfen where Note between 1 and 2;</pre>	<pre>insert into prüfen values(29555, 5001, 2137, 1);</pre>

- Problem lässt sich dadurch lösen, dass man zusätzlich zu den Tupeln auch den Zugriffsweg, auf den man zu den Objekten gelangt, sperrt.
- Wenn also ein Sekundärindex auf Note existiert, würde der Indexbereich [1;2] für  $T_1$  mit einer S-Sperre belegt werden.
- Wenn jetzt  $T_2$  versucht, das Tupel (29555, 5001, 2137, 1) in Tabelle *prüfen* einzufügen, wird die  $T_2$  blockiert.
- Zusätzlich müssen die Tupel gesperrt werden, da der Zugriff möglicherweise nicht über den Index erfolgt...

## Optimistische Synchronisation

**Grundidee: Erst beim Commit wird entschieden, ob es einen Konflikt gegeben hat**

### 1. Lesephase

- Alle Operationen der TA einschliesslich Änderungsoperationen werden ausgeführt
- Geänderte Werte werden noch nicht in die Datenbasis eingebracht sondern TA-lokal als Kopie gespeichert
- alle Änderungsoperationen werden zunächst auf diesen lokalen Variablen durchgeführt

### 2. Validierungsphase

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

## Optimistische Synchronisation

### 3. Schreibphase

- Die Änderungen derjenigen TAs, bei denen die Validierung positiv verlaufen ist, werden in die Datenbasis eingebracht. Sonst wird die TA zurückgesetzt.

### Bemerkungen

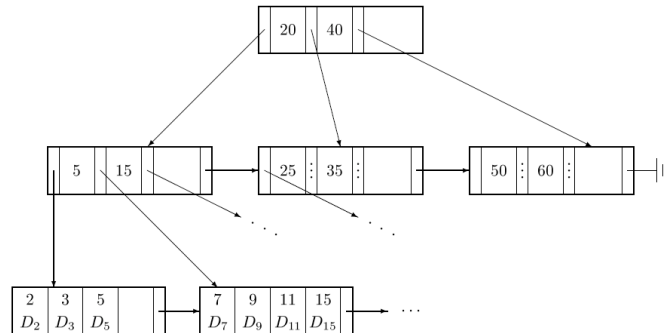
- TAs, die zurückgesetzt werden, haben keinen Effekt auf die Datenbasis, deswegen gibt es auch kein kaskadierendes Zurücksetzen.
- In der Datenbasis gibt es also nur Änderungen erfolgreicher TAs.
- Optimistische Synchronisation eignet sich insbesondere für DB-Anwendungen mit überwiegend Lese-TAs.

## Validierung bei optimistischer Synchronisation

- Vereinfachende Annahme: es ist immer nur eine TA gleichzeitig in der Validierungsphase
- Wir wollen eine Transaktion  $T_j$  validieren.
- Die Validierung ist erfolgreich, falls für alle älteren Transaktionen  $T_a$  - also solche die bereits ihre Validierung abgeschlossen haben - eine der beiden folgenden Bedingungen gilt:
  - $T_a$  war zum Beginn von  $T_j$  bereits abgeschlossen - einschliesslich der Schreibphase.
  - Die Menge der von  $T_a$  geänderten Datenelemente, genannt *WriteSet* ( $T_a$ ), enthält keine Elemente der Menge der gelesenen Datenelemente von  $T_j$ , genannt *ReadSet*( $T_j$ ). Es muss also gelten:

$$\text{WriteSet}(T_a) \cap \text{ReadSet}(T_j) = \emptyset$$

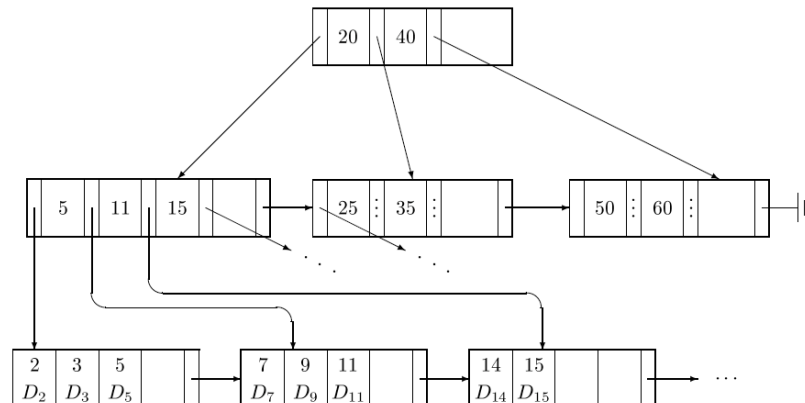
## Synchronisation von Indexstrukturen



- Strenges 2PL würde kompletten Pfad auf dem Weg zum Tupel sperren
- Effekt: zu grosse Teile des B<sup>+</sup>-Baums unnötigerweise gesperrt
- Idee: füge rechts-Verweise auf den (Nicht-Blatt-) Knoten ein
- Schreiboperation: nur Knoten sperren, nicht den Pfad darüber

## Synchronisation von Indexstrukturen

- Zwei parallele Operationen: probe(15) & insert(14)



- probe erreicht Blattknoten [7;15] und wird dann angehalten
- insert spaltet Blattknoten [7;15] in [7;11] und [14;15]
- probe rechnet weiter: Schlüssel 15 ist nicht mehr auf dem Blatt [7;11] !
- probe muss nach rechts laufen! (kann auch Nicht-Blatt-Knoten betreffen)

## Transaktionsverwaltung in SQL

### set transaction

[read only, | read write,]

[isolation level

read uncommitted, |

read committed, |

repeatable read, |

serializable,

[diagnostics size ...,]

### ▪ read uncommitted

- liest beliebig inkonsistente Datenbestände
- darf nur für read-only TAs spezifiziert werden
- für browsing geeignet
- TA behindert parallele Ausführung anderer Transaktionen nicht, da TA keine Sperren benötigt

## Transaktionsverwaltung in SQL

### ▪ read committed

- liest nur festgeschriebene Werte
- TA kann innerhalb der Transaktion möglicherweise unterschiedliche Zustände von Werten sehen:  
**non-repeatable read-Problematik**

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

### ▪ repeatable read

- non-repeatable read ist ausgeschlossen
- Phantomproblem möglich, beispielsweise durch eine parallele Änderungs-Transaktion, die dazu führt, dass ein Tupel jetzt ein Selektionsprädikat erfüllt, das es vorher nicht erfüllt hat

### ▪ serializable

- Serialisierbarkeit gefordert
- Dies ist der default.

# Nächste Woche

Ausblick

DataSpace Management Systems

Semester- und Masterarbeiten in der DBIS-Gruppe

