Kapitel 10

Mehrbenutzersynchronisation

Mehrbenutzersynchronisation

- Alle TAs strikt seriell (also nacheinander) auszuführen ist sicher, aber langsam
- Oft werden Systemressourcen nicht voll ausgenutzt, da eine TA auf Plattenzugriff oder Benutzereingabe wartet
- Diese TA blockiert dann alle anderen TAs
- Um Systemressourcen auszunutzen bietet sich Nebenläufigkeit an

Mehrbenutzersynchronisation(2)

- Nicht abgesicherte Nebenläufigkeit kann aber zu folgenden Problemen führen:
  - lost update
  - dirty read
  - non-repeatable read
  - phantom problem

Lost Update

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$r_2(x)$
$w_1(x)$	$\hookleftarrow$
$\hookrightarrow$	$w_2(x)$
commit	$\hookleftarrow$
$\hookrightarrow$	commit

Das Ergebnis der Transaktion T_1 ist verlorengegangen!

Dirty Read

T_1	T_2
bot	
$\hookrightarrow$	bot
	$r_2(x)$
	$w_2(x)$
$r_1(x)$	$\hookleftarrow$
$w_1(y)$	
commit	
$\hookrightarrow$	abort

T_1 liest einen Wert für x der so nicht gültig ist!

Non-Repeatable Read

T_1	T_2
bot	
$r_1(x)$	
$\hookrightarrow$	bot
	$w_2(x)$
	commit
$r_1(x)$	$\hookleftarrow$
...	

T_1 liest x zweimal mit verschiedenem Ergebnis!

Phantom Problem

T_1	T_2
bot	
select count(*) from R;	
$\hookrightarrow$	bot
	insert into R ...;
	commit
select count(*) from R;	$\hookleftarrow$
...	

T_1 findet ein weiteres Tupel beim Abarbeiten der zweiten Anfrage!

Vermeidung der Probleme

- Im Idealfall sollten alle diese Probleme vermieden werden
- Man muß dabei allerdings einen Kompromiß zwischen Performanz und Genauigkeit schließen
- Je mehr Sicherheit, desto langsamer wird die Ausführung
- Über die *Isolation Levels* kann man DBMS mitteilen, welche Sicherheit erwünscht ist

Transaktionen und SQL

- Festsetzen von Eigenschaften einer TA:

  set transaction *Stufe, Zugriffsmodus*

- Folgende Stufen für den Isolation Level sind möglich:
  - read uncommitted
  - read committed
  - repeatable read
  - serializable
- Mögliche Zugriffsmodi:
  - read only
  - read write

Transaktionen und SQL(2)

	lost update	dirty read	nonrep. read	phant. probl.
read uncommit.	√			
read commited	√	√		
repeat. read	√	√	√	
serializable	√	√	√	√

Transaktionen und SQL(3)

- "read only" sagt dem DBMS, daß eine TA nur Leseoperationen enthält
- Das hat Auswirkungen auf die Performanz
- Nebenläufiges Ausführen von TAs die nur lesen ist unkritisch, d.h. beliebig vieler solcher TAs können völlig uneingeschränkt parallel laufen
- Erst wenn eine TA dazukommt, die auch schreibt müssen Vorkehrungen getroffen werden

Transaktionen und SQL(4)

- Befehl zum Markieren eines Transaktionsbeginns

  start transaction;

- Befehl zur erfolgreichen Beendigung

  commit [work];

- Befehl zum Abbruch

  rollback [work];

Was macht ein DBMS?

- Um Lösungsansätze besser verstehen zu können, wird das Problem zunächst etwas formaler betrachtet
- Danach werden Lösungen vorgestellt, die in DBMS eingesetzt werden

Formale Definition einer TA

- Operationen einer TA T_i
  - $r_i(A)$: Lesen des Datenobjekts A
  - $w_i(A)$: Schreiben des Datenobjekts A
  - a_i: Abbruch
  - c_i: erfolgreiche Beendigung

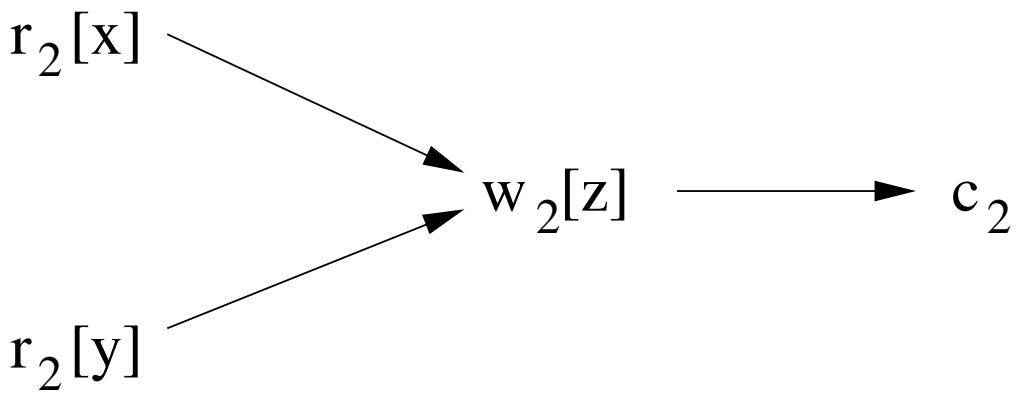
  - *bot*: begin of transaction (implizit)

Formale Definition einer TA(2)

- Eine TA T_i ist eine partielle Ordnung von Operationen mit der Ordnungsrelation $<_i$ so daß:
  - $T_i \subseteq \{r_i[x], w_i[x] \mid x \text{ ist ein Datenobjekt}\}$ $\cup\{a_i, c_i\}$
  - $a_i \in T_i$, gdw. $c_i \notin T_i$
  - Sei t gleich a_i oder c_i, dann gilt für jede andere Operation p_i: $p_i <_i t_i$
  - Falls $r_i[x]$ und $w_i[x] \in T_i$, dann gilt entweder $r_i[x] <_i w_i[x]$ oder $w_i[x] <_i r_i[x]$

Darstellung

- Transaktionen werden oft als gerichtete azyklische Graphen (DAGs) dargestellt:



- $r_2[x] <_2 w_2[z]$, $w_2[z] <_2 c_2$, $r_2[x] <_2 c_2$, $r_2[y] <_2 w_2[z]$, $r_2[y] <_2 c_2$
- Transitive Beziehungen sind im Graph implizit enthalten

Historien (Schedules)

- Mehrere TAs können nebenläufig ausgeführt werden
- Dies wird durch eine *Historie* (Schedule) beschrieben
- Eine Historie gibt an, wie Operationen aus verschiedenen TAs relativ zueinander ausgeführt werden
- Da verschiedene Operationen parallel ausgeführt werden können, ist eine Historie eine partielle Ordnung
-

Konfliktoperationen

- Operationen die in Konflikt miteinander stehen dürfen nicht parallel ausgeführt werden
- Zwei Operationen stehen in Konflikt miteinander, wenn beide auf dem gleichen Datenobjekt arbeiten und mindestens eine davon eine Schreiboperation ist

	T_i	
T_j	$r_i[x]$	$w_i[x]$
$r_j[x]$		$\neg$
$w_j[x]$	$\neg$	$\neg$

Definition von Historien

- Sei $T = \{T_1, T_2, \ldots, T_n\}$ eine Menge von Transaktionen
- Eine Historie H über T ist eine partielle Ordnung mit der Ordnungsrelation $<_H$, so daß
  - $H = \bigcup_{i=1}^{n} T_i$
  - $<_H \supseteq \bigcup_{i=1}^{n} <_i$
  - Für zwei beliebige Operationen $p, q \in H$ die in Konflikt miteinander stehen gilt: entweder $p <_H q$ oder $q <_H p$

Beispiel einer Historie

$$
\begin{array}{lccccc}
 & & r_2[x] \rightarrow & w_2[y] \rightarrow & w_2[z] \rightarrow & c_2 \\
 & & \uparrow & \uparrow & \uparrow & \\
H = & r_3[y] \rightarrow & w_3[x] \rightarrow & w_3[y] \rightarrow & w_3[z] \rightarrow & c_3 \\
 & & \uparrow & & & \\
 & r_1[x] \rightarrow & w_1[x] \rightarrow & c_1 & &
\end{array}
$$

(Konflikt-)Äquivalenz

- Zwei Historien H und H' sind *(konflikt-)äquivalent* ($H \equiv H'$), wenn:
  - Sie enthalten die gleichen Mengen von TAs (samt allen dazugehörigen Operationen)
  - Sie ordnen die Konfliktoperationen der nicht abgebrochenen TAs in der gleichen Art und Weise an
- Die Idee dabei ist, das berechnete Endergebnis nicht zu verändern

Beispiel

$$
\begin{aligned}
& r_1[x] \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv \quad & r_1[x] \rightarrow r_2[z] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[y] \rightarrow c_2 \\
\equiv \quad & r_2[z] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[y] \rightarrow c_2 \rightarrow c_1 \\
\not\equiv \quad & r_2[z] \rightarrow r_1[x] \rightarrow w_2[y] \rightarrow w_1[y] \rightarrow c_2 \rightarrow c_1
\end{aligned}
$$

Serialisierbarkeit

- Da serielle Historien sicher sind, ist es wünschenswert Historien mit ähnlichen Eigenschaften zu haben
- Insbesondere möchte man eine Historie haben die äquivalent zu einer seriellen Historie ist
- Eine solche Historie nennt man *serialisierbar*

Serialisierbarkeit(2)

- Präzise Definition:
  - Die *abgeschlossene Projektion* $C(H)$ einer Historie H enthält nur die erfolgreich abgeschlossenen TAs
  - Eine Historie H ist serialisierbar, wenn $C(H)$ äquivalent zu einer seriellen Historie H_s ist

Serialisierbarkeitstheorem

- Wie überprüft man die Serialisierbarkeit?
- Eine Historie ist genau dann serialisierbar, wenn ihr *Serialisierbarkeitsgraph* $SG(H)$ azyklisch ist

Serialisierbarkeitsgraph

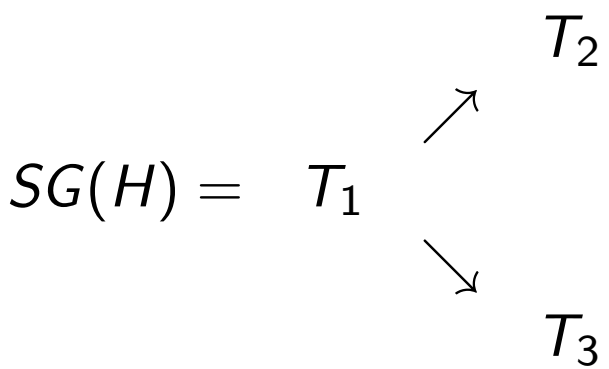
- Der Serialisierbarkeitsgraph $SG(H)$ einer Historie $H = \{T_1, \ldots, T_n\}$ ist ein gerichteter Graph mit folgenden Eigenschaften:
  - Die Knoten sind die erfolgreich abgeschlossenen TAs aus H
  - Eine Kante zwischen zwei TAs T_i und T_j wird eingetragen, wenn es zwei Konfliktoperationen p_i und q_j gibt und $p_i <_H q_j$

Beispiel

- Historie H

$$H = w_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow r_3[y] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_3[y] \rightarrow c_3$$

- $SG(H)$



Beispiel(2)

- H ist serialisierbar
- Mögliche Ordnungen

$$H_s^1 = T_1 \mid T_2 \mid T_3$$
$$H_s^2 = T_1 \mid T_3 \mid T_2$$
$$H \equiv H_s^1 \equiv H_s^2$$

Beispiel(3)

$$H = \begin{array}{ccccccc} r_1[x] & \rightarrow & w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 \\ & & \uparrow & & \uparrow & & \\ & \swarrow & r_2[x] & \rightarrow & w_2[y] & \rightarrow & c_2 \\ & & \downarrow & & & & \\ r_3[x] & \rightarrow & w_3[x] & \rightarrow & c_3 & & \end{array}$$

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

Beispiel(4)

- H ist serialisierbar
- Mögliche Ordnung

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

Beispiel(5)

$$H = \begin{array}{ccccc} w_1[x] & \rightarrow & w_1[y] & \rightarrow & c_1 \\ \uparrow & & \downarrow & & \\ r_2[x] & \rightarrow & w_2[y] & \rightarrow & c_2 \end{array}$$

$$SG(H) = \quad T_1 \quad \leftrightarrows \quad T_2$$

- H ist nicht serialisierbar

Weitere Eigenschaften

- Für TAs sind weitere Eigenschaften wünschenswert:
  - Rücksetzbarkeit (Recoverability)
  - Vermeidung kaskadierenden Rücksetzens (avoiding cascading aborts: ACA)
  - Striktheit (strictness)

Weitere Eigenschaften(2)

- Zuerst müssen wir Schreib-/Leseabhängigkeiten (reads-from relationship) definieren
- Eine TA T_i liest (Datenobjekt x) von TA T_j, wenn
  - $w_j[x] < r_i[x]$
  - $a_j \not< r_i[x]$
  - Falls ein $w_k[x]$ existiert mit $w_j[x] < w_k[x] < r_i[x]$, dann $a_k < r_i[x]$
- Eine TA kann auch von sich selbst lesen

Rücksetzbarkeit

- Eine Historie ist *rücksetzbar*, wenn folgendes gilt
  - Immer wenn eine TA T_i von einer anderen TA T_j liest ($i \neq j$) und $c_i \in H$, dann $c_j < c_i$
- Die TAs müssen eine bestimmte Commit-Reihenfolge einhalten
- Bei nicht rücksetzbaren Historien können Probleme mit dem C und D der ACID-Eigenschaften auftreten

Rücksetzbarkeit(2)

$$H = w_1[x]\ r_2[x]\ w_2[y]\ c_2\ a_1$$

- H ist nicht rücksetzbar
- Die Konsequenzen sind:
  - Wenn Ergebnis von T_2 so stehen bleibt, dann haben wir inkonsistente Daten (T_2 hat Daten von einer abgebrochenen TA gelesen)
  - Wenn wir T_2 zurücksetzen, dann nehmen wir Änderungen einer fest zugesicherten TA zurück

Kaskadierendes Rücksetzen

Schritt	T_1	T_2	T_3	T_4	T_5
0.	$\cdots$				
1.	$w_1[x]$				
2.		$r_2[x]$			
3.		$w_2[y]$			
4.			$r_3[y]$		
5.			$w_3[z]$		
6.				$r_4[z]$	
7.				$w_4[v]$	
8.					$r_5[v]$
9.	a_1 (**abort**)				

Kaskadierendes Rücksetzen(2)

- Eine Historie *vermeidet kaskadierendes Rücksetzen*, wenn folgendes gilt
  - Immer wenn eine TA T_i von einer anderen TA T_j liest ($i \neq j$), dann $c_j < r_i[x]$
- Es darf nur von bereits erfolgreich abgeschlossenen TAs gelesen werden

Striktheit

- Eine Historie ist *strikt*, wenn folgendes gilt
  - Bei zwei Operationen $w_j[x] < o_i[x]$ (mit $o_i[x] = r_i[x]$ oder $w_i[x]$ gilt entweder $a_j < o_i[x]$ oder $c_j < o_i[x]$
- Nur von bereits erfolgreich abgeschlossenen TAs darf gelesen oder dürfen Datenobjekte überschrieben werden

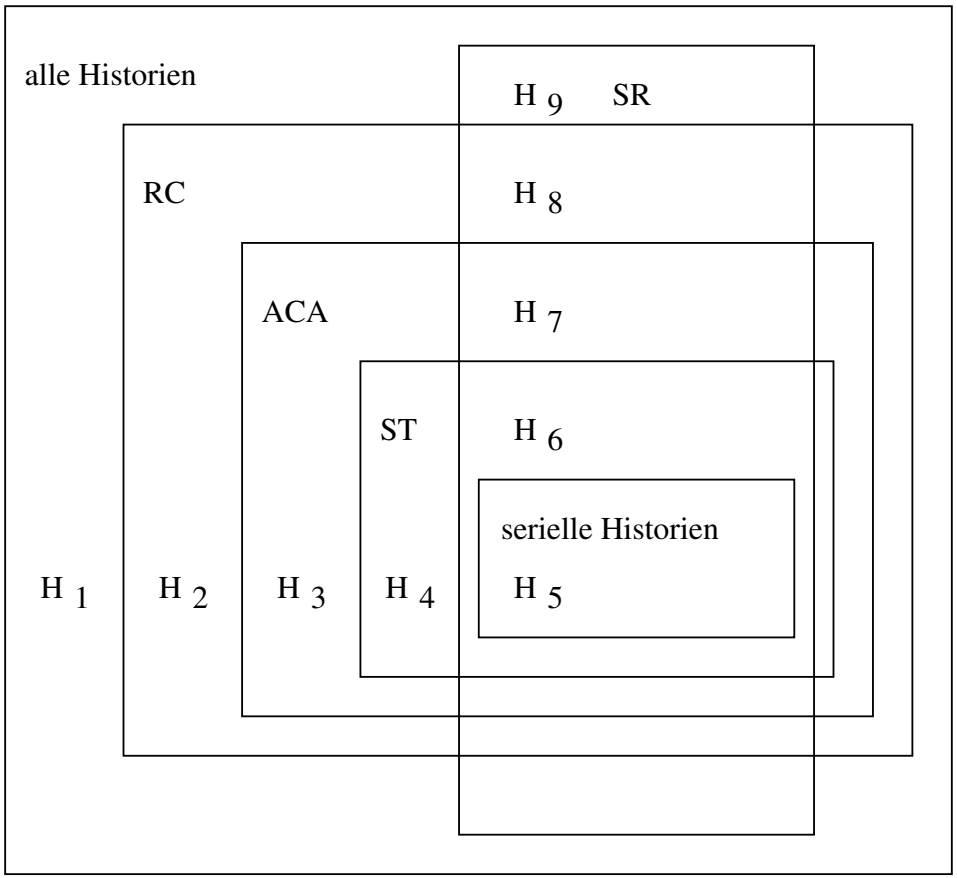
Striktheit(2)

- Nur bei strikten Historien darf physische Protokollierung beim Recovery angewendet werden

$$
\begin{array}{ll}
x = 0 & \\
w_1[x, 1] & \text{before image von } T_1\text{: } 0 \\
x = 1 & \\
w_2[x, 2] & \text{before image von } T_2\text{: } 1 \\
x = 2 & \\
a_1 & \\
c_2 &
\end{array}
$$

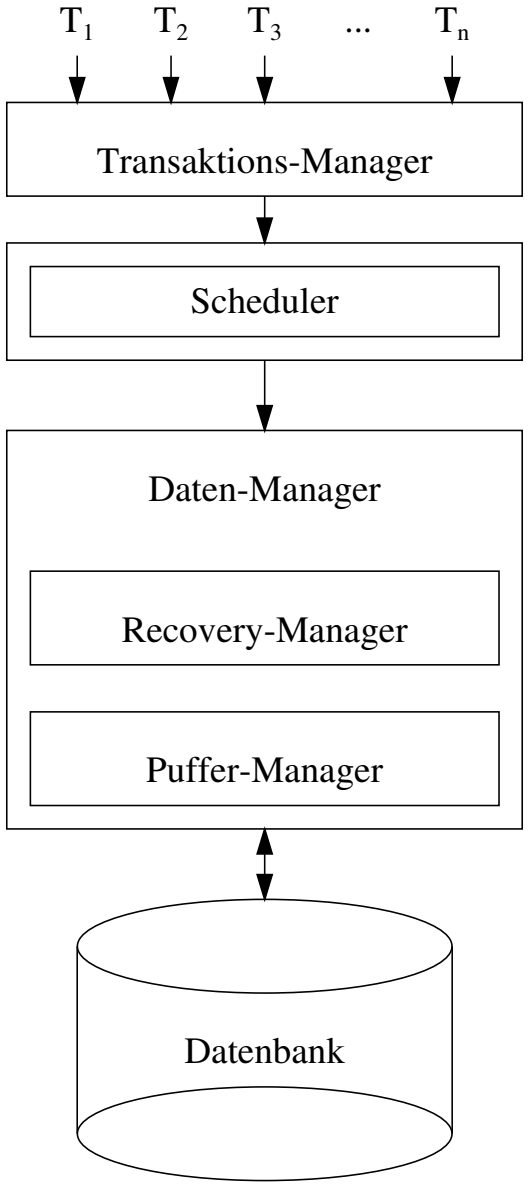
Bei Abbruch von T_1 wird x fälschlicherweise auf 0 gesetzt

Einordnung



SR: serialisierbar, RC: rücksetzbar, ACA: vermeidet kaskadierendes Rücksetzen, ST: strikt

Datenbank-Scheduler



Datenbank-Scheduler(2)

- Ein *Scheduler* ist ein Programm, das die eingehenden Operationen ordnet und für eine serialisierbare und rücksetzbare Historie sorgt
- Mehrere Möglichkeiten nach Entgegennahme einer Operation:
  - (Sofort) ausführen
  - Zurückweisen
  - Verzögern

Datenbank-Scheduler(3)

- Es existieren zwei grobe Strategien:
  - Pessimistisch
  - Optimistisch

Pessimistische Scheduler

- Scheduler verzögert entgegengenommene Operationen
- Wenn mehrere Operationen da sind, legt Scheduler möglichst geschickte Reihenfolge fest
- Wichtigster Vertreter: Sperrbasierter Scheduler (in der Praxis weit verbreitet)

Optimistische Scheduler

- Scheduler schickt entgegengenommene Operationen möglichst schnell zur Ausführung,
- Muß später eventuell "Schaden" reparieren
- Wichtigster Vertreter: Zeitstempelbasierter Scheduler

Sperrbasierte Synchronisation

- Hauptidee relativ einfach:
  - Jedes Datenobjekt hat eine zugehörige Sperre
  - Bevor eine TA T_i zugreifen darf, muß sie Sperre anfordern
  - Falls eine andere TA T_j Sperre hält, bekommt T_i die Sperre nicht und muß warten, bis T_j die Sperre freigegeben hat
  - Nur eine TA kann Sperre halten und auf Datenobjekt zugreifen
- Wie garantiert man Serialisierbarkeit?

Zwei-Phasen-Sperrprotokoll

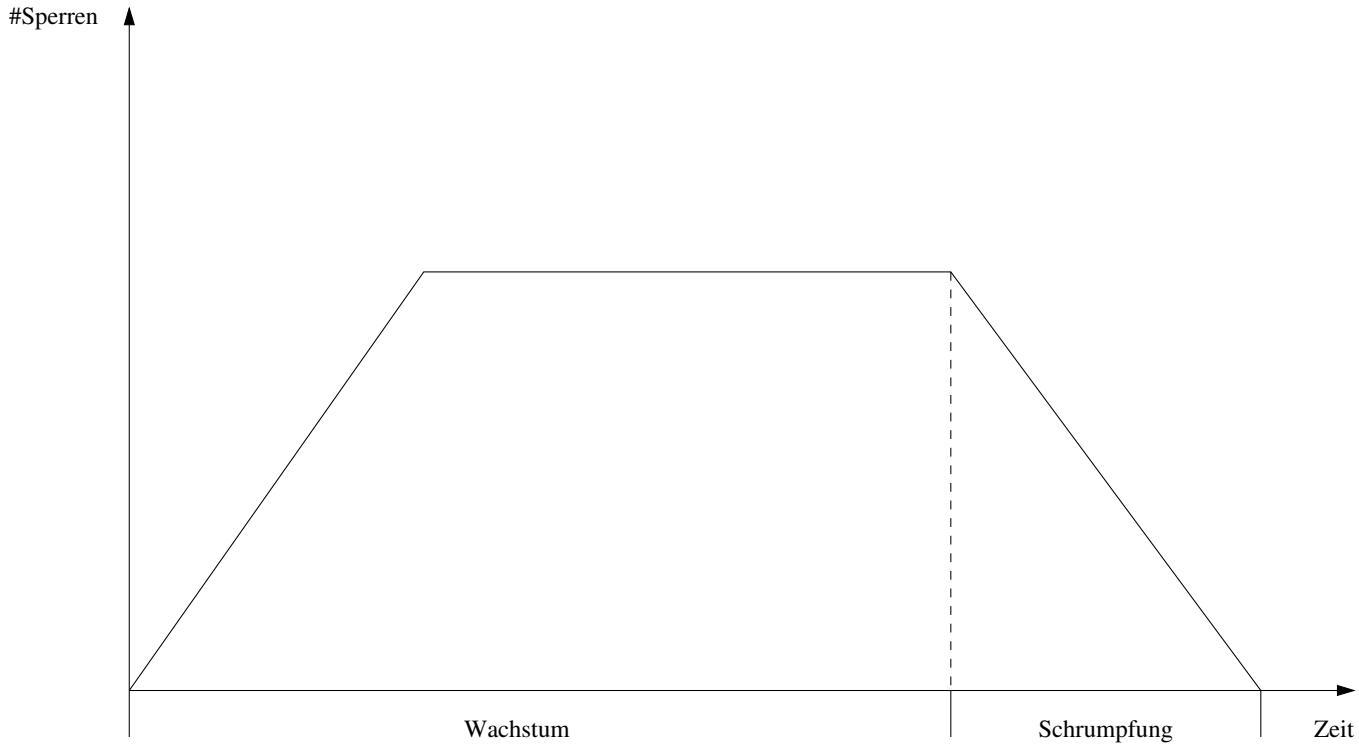
- Abgekürzt durch 2PL
- Zwei Sperrmodi:
  - S (shared, read lock, Lesesperre)
  - X (exclusive, write lock, Schreibsperre)
  - Verträglichkeitsmatrix (auch Kompatibilitätsmatrix genannt):

	gehaltene Sperre		
angeford. Sp.	keine	S	X
S	$\surd$	$\surd$	–
X	$\surd$	–	–

Definition

- Jedes Objekt, das von einer TA benutzt werden soll, muß vorher entsprechend gesperrt werden
- Eine TA kann eine Sperre die sie hält nicht noch einmal anfordern
- Wenn eine Sperre nicht gewährt werden kann (nach Matrix), wird TA in Warteschlange eingereiht
- Eine TA darf nach der ersten Freigabe einer Sperre keine weitere anfordern (es gibt zwei Phasen)
- Bei Transaktionsende muß eine TA alle Sperren zurückgeben

Zwei Phasen



- Wachstumsphase: es werden Sperren angefordert, aber keine freigegeben
- Schrumpfungsphase: es werden Sperren freigegeben, aber keine angefordert

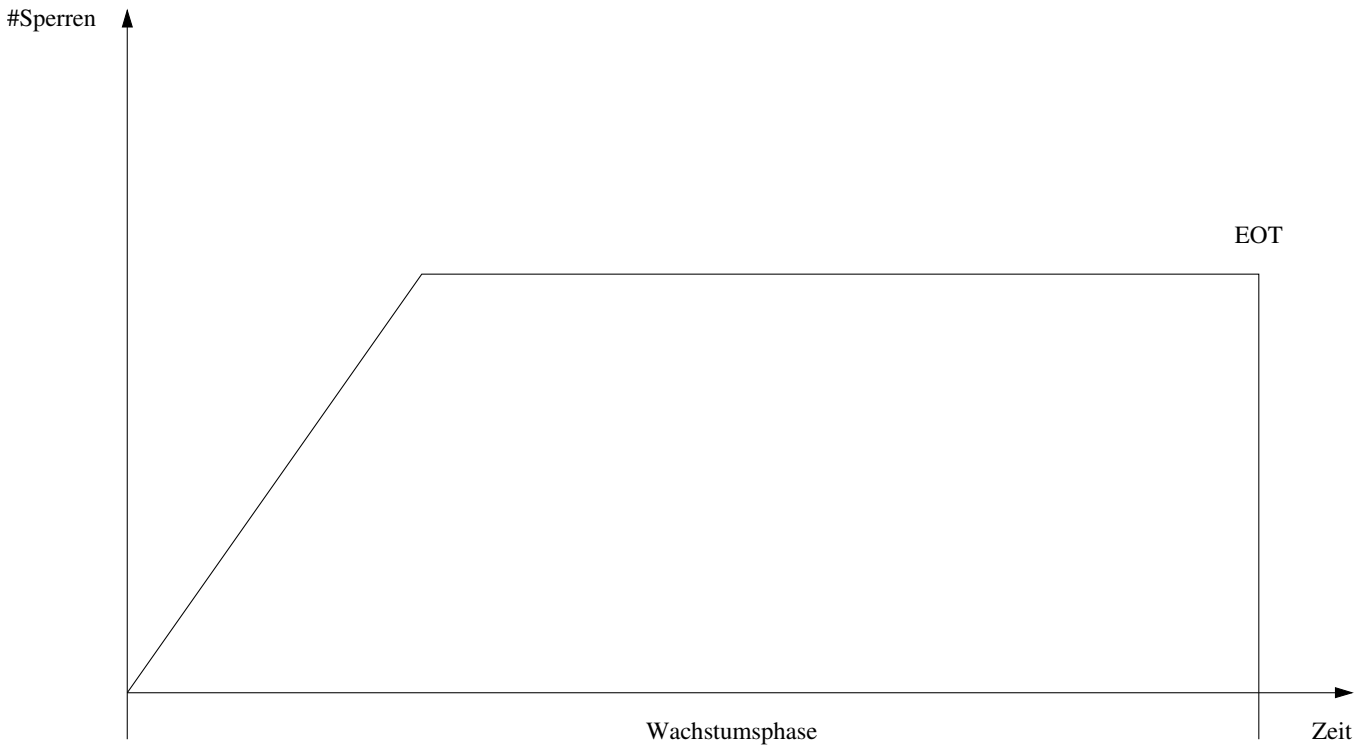
Verzahnung nach 2PL

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**$[x]$		
3.	$r[x]$		
4.	$w[x]$		
5.		**BOT**	
6.		**lockS**$[x]$	T_2 muss warten
7.	**lockX**$[y]$		
8.	$r[y]$		
9.	**unlockX**$[x]$		T_2 wecken
10.		$r[x]$	
11.		**lockS**[y]	T_2 muss warten
12.	$w[y]$		
13.	**unlockX**$[y]$		T_2 wecken
14.		$r[y]$	
15.	**commit**		
16.		**unlockS**$[x]$	
17.		**unlockS**$[y]$	
18.		**commit**	

Strenges 2PL

- 2PL schließt kaskadierendes Rücksetzen nicht aus
- Erweiterung zum *strengen* 2PL:
  - alle Sperren werden bis zum Ende der Transaktion gehalten
  - damit ist kaskadierendes Rücksetzen ausgeschlossen (die erzeugten Schedules sind sogar strikt)

Strenges 2PL(2)



Verklemmungen (Deadlocks)

- Beispiel für ein Deadlock:

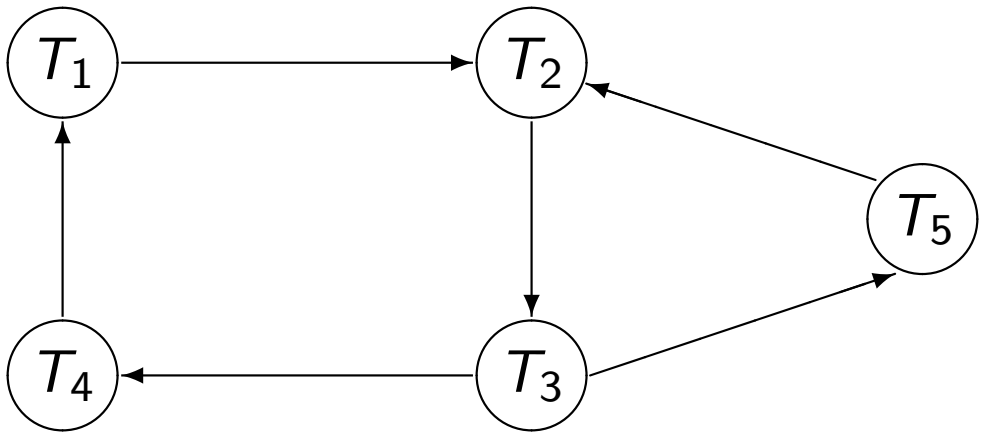
T_1	T_2
bot	
$\text{lockX}_1(a)$	
$w_1(a)$	
$\hookrightarrow$	bot
	$\text{lockS}_2(b)$
	$r_2(b)$
$\text{lockX}_1(b)$	$\hookleftarrow$
$\hookrightarrow$	$\text{lockS}_2(a)$

Deadlocks erkennen

- Keine TA soll "ewig" auf eine Sperre warten
- Eine Strategie zum Erkennen von Deadlocks ist Time-Out
  - Richtige Zeitdauer zu finden ist problematisch
- Präzise Methode benutzt Wartegraphen
  - Knoten sind TAs, Kanten sind Wartet-auf-Beziehungen
  - Wenn Graph Zyklen aufweist, liegt ein Deadlock vor

Wartegraph

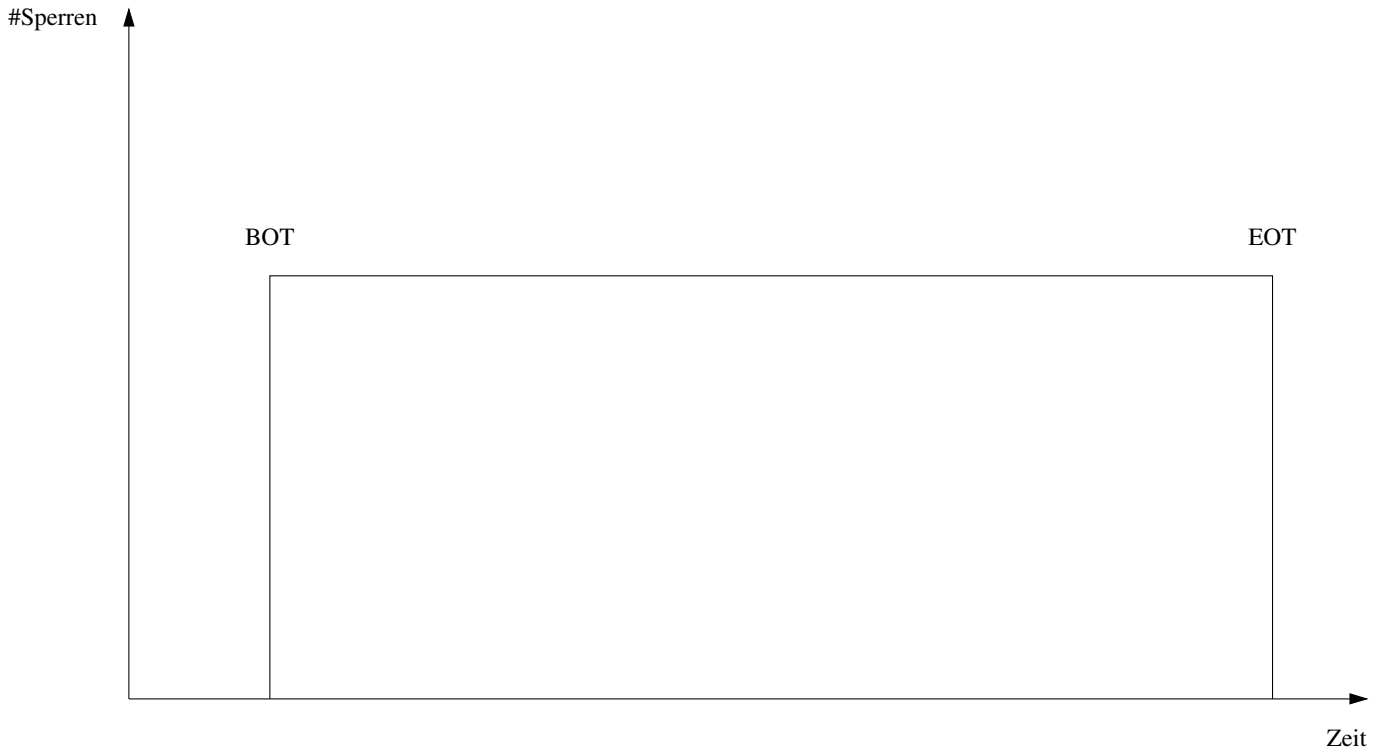
- Beispiel



- Wartegraph hat Zyklen, d.h. Deadlock liegt vor
- Zyklen können hier durch Zurücksetzen von T_2 oder T_3 aufgelöst werden

Deadlock-Vermeidung

- Deadlocks können durch *Preclaiming* vermieden werden
- Preclaiming bedeutet, daß alle Sperren zu Beginn einer TA angefordert werden
- In der Praxis unrealistisch



Deadlock-Vermeidung(2)

- Time-Out-Verfahren ist oft zu vorsichtig, z.T. werden TAs abgebrochen, wenn nur der Verdacht auf ein Deadlock besteht
- Eine andere Methode besteht darin zum Zeitpunkt wenn T_i eine Sperre anfordert, die von T_j gehalten wird, zu entscheiden
  - TAs bekommen Prioritäten zugewiesen
  - Wenn Priorität von T_i höher ist, darf T_i warten
  - Wenn Priorität von T_i kleiner ist, wird T_i abgebrochen

Deadlock-Vermeidung(3)

- Durch Prioriäten wird vermieden, daß durch ein Warten ein Deadlock entstehen kann
- Prioritätenvergabe muß umsichtig erfolgen
  - Wenn eine abgebrochene TA T_i beim Neutstart ständig niedrige Prioritäten erhält, können sich immer TAs mit höheren Prioritäten "vordrängeln"
  - T_i kommt nie zum Zug, wir haben kein Deadlock, aber ein *Livelock*

Deadlock-Vermeidung(4)

- Vermeidung von Deadlocks und Livelocks: Verwendung von Zeitstempeln als Prioritäten
- Zeitstempel sind eindeutig und wachsen monoton mit der Zeit
- Eine TA bekommt beim ersten Aufruf einen Zeitstempel *ts* zugewiesen, beim Neustart behält sie den alten Zeitstempel
- Je älter der Zeitstempel, desto höher die Priorität
- Irgendwann hat eine immer wieder abgebrochene TA den ältesten Zeitstempel, Livelocks werden verhindert

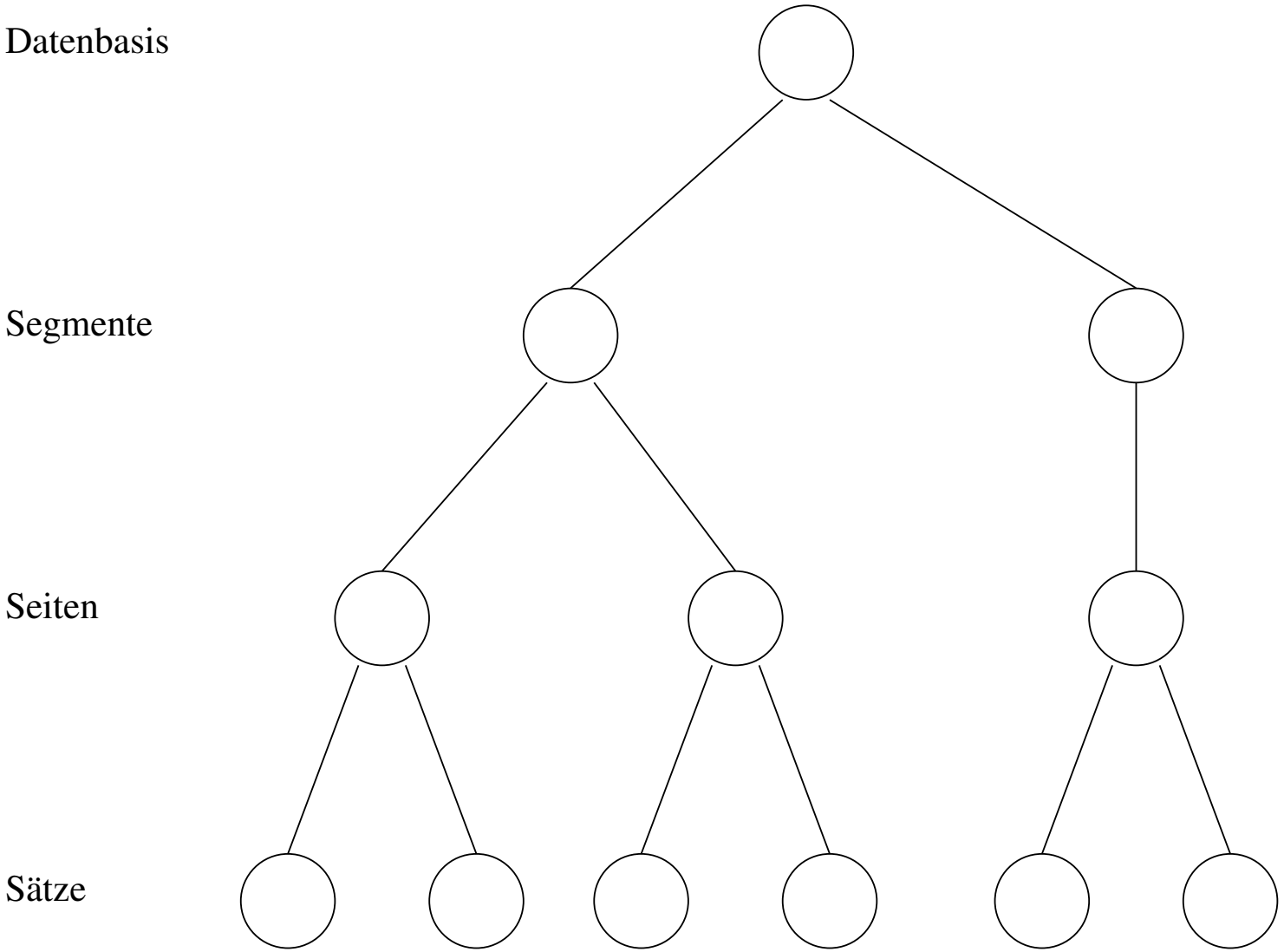
Deadlock-Vermeidung(5)

- Angenommen T_j hält Sperre, T_i fordert sie an
- Im Zeitstempelverfahren kann ein Scheduler nun zwei verschiedene Strategien fahren
  - Wait-Die:
    Falls $ts(T_i) < ts(T_j)$,
    dann wartet T_i, sonst bricht T_i ab
  - Wound-Wait:
    Falls $ts(T_i) < ts(T_j)$,
    dann bricht T_j ab, sonst wartet T_i

Phantom-Problem

- Mit (strengem) 2PL haben wir alle am Anfang des Kapitels angesprochenen Probleme gelöst, außer des Phantom-Problems
- Phantom-Problem läßt sich mit Sperren auf Datenobjekten nicht lösen, da keine Sperren auf nichtexistenten Datenobjekten angefordert werden können
- Lösung des Problems durch *hierarchische Sperrgranulate* (multi-granularity locking: MGL)

MGL



Erweiterte Sperrmodi für MGL

- *S* (shared): für Leser
- *X* (exclusive): für Schreiber
- *IS* (intention share): für beabsichtigtes Lesen weiter unten in der Hierarchie
- *IX* (intention exclusive): für beabsichtigtes Schreiben weiter unten in der Hierarchie

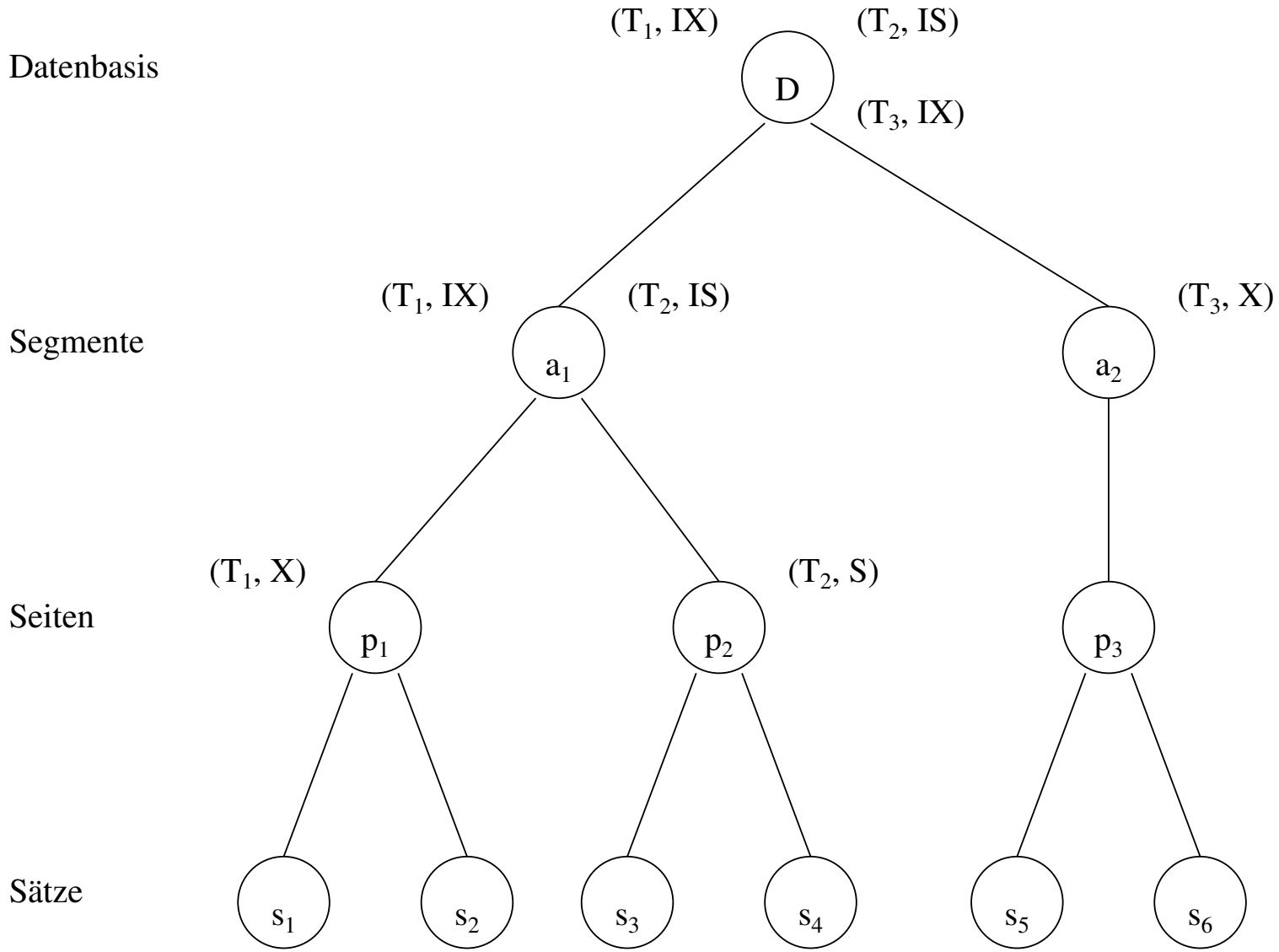
Kompatibilitätsmatrix

	gehaltene Sperre				
angef. Sp.	keine	*S*	*X*	*IS*	IX
S	√	√	–	√	–
X	√	–	–	–	–
IS	√	√	–	√	√
IX	√	–	–	√	√

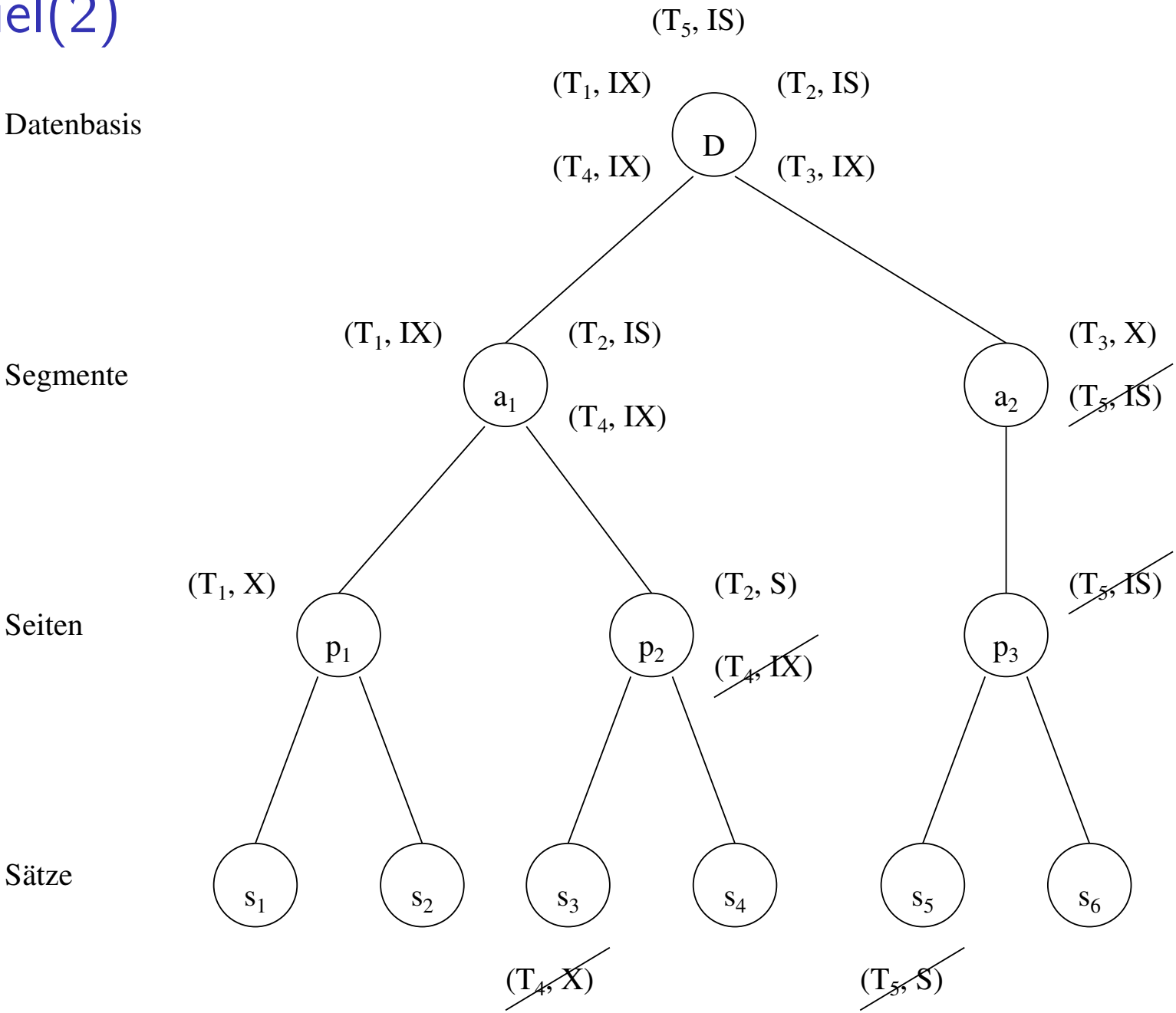
Sperrprotokoll

- Sperren werden in der Hierarchie von oben nach unten angefordert
  - Für eine *S* oder *IS* Sperre müssen alle Vorgänger in der Hierarchie im *IS* oder *IX* Modus gesperrt sein
  - Für eine *X* oder *IX* Sperre müssen alle Vorgänger in der Hierarchie im *IX* Modus gehalten werden
- Sperren werden von unten nach oben wieder freigegeben (Sperre wird nur freigegeben, wenn auf keinem Nachfolger des Knotens noch eine Sperre gehalten wird)

Beispiel


Datenbasis
Segmente
Seiten
Sätze
(T1, IX)
(T2, IS)
D
(T3, IX)
(T1, IX)
(T2, IS)
(T3, X)
a1
a2
(T1, X)
(T2, S)
p1
p2
p3
s1
s2
s3
s4
s5
s6

Beispiel(2)


Datenbasis
Segmente
Seiten
Sätze
(T5, IS)
(T1, IX)
(T2, IS)
(T4, IX)
(T3, IX)
D
(T1, IX)
(T2, IS)
(T4, IX)
(T3, X)
(T5, IS)
a1
a2
(T1, X)
(T2, S)
(T4, IX)
(T5, IS)
p1
p2
p3
s1
s2
s3
s4
s5
s6
(T4, X)
(T5, S)

Beispiel(3)

- TAs T_4 und T_5 sind blockiert
- Hier noch kein Deadlock, aber im einfachen MGL-Protokoll sind auch Deadlocks möglich

Zeitstempelbasierte Verfahren

- Neben den sperrbasierten Protokollen gibt es noch eine weitere große Klasse an Protokollen, die zeitstempelbasierte Synchronisation
- Transaktionsmanager weist jeder TA einen eindeutigen Zeitstempel zu
- Jede Operation der TA bekommt diesen Zeitstempel

Zeitstempel

- Ein Scheduler benutzt die Zeitstempel um in Konflikt stehende Operationen zu ordnen:
  - Angenommen $p_i[x]$ und $q_j[x]$ stehen in Konflikt miteinander
  - $p_i[x]$ wird vor $q_j[x]$ ausgeführt, gdw. der Zeitstempel von T_i älter als der Zeitstempel von T_j ist

Zeitstempel(2)

- Scheduler speichert zu jedem Datenobjekt x den Zeitstempel der letzten auf x ausgeführten Operation
- Das wird für jeden Operationstypen q gemacht: max-q-scheduled(x)
- Wenn Scheduler eine Operation p bekommt, wird ihr Zeitstempel mit allen max-q-scheduled(x) verglichen, mit denen p in Konflikt steht
- Wenn der Zeitstempel von p älter als ein max-q-scheduled(x) ist, wird p zurückgewiesen (und TA abgebrochen)
- Ansonsten wird p ausgeführt und max-p-scheduled(x) aktualisiert

Weitere Eigenschaften

- Einfaches Zeitstempelverfahren erzeugt u.U. nicht rücksetzbare Schedules
- Rücksetzbarkeit kann dadurch garantiert werden, daß TAs in Zeitstempelreihenfolge committen
- Solange noch TAs laufen, von denen eine TA T_i gelesen hat, wird ein commit von T_i verzögert

Probleme von Zeitstempeln

- Zeitstempelbasierte Synchronisation wird in der Praxis kaum eingesetzt
- Phantom-Problem wird nicht gelöst
- Jede Operation wird praktisch zur Schreiboperation, da immer die max-q-scheduled(x)-Felder aktualisiert werden müssen

Zusammenfassung

- Mehrbenutzersynchronisation gehört mit zu den wichtigsten Funktionen eines DBMS
- Normalerweise bleibt dies den Benutzern verborgen, aber über die Einstellung der Isolation-Levels kann in die Qualität dieser Synchronisation eingegriffen werden
- Die zwei bekanntesten Verfahren:
  - Sperrbasierte Synchronisation
  - Zeitstempelbasierte Synchronisation