Informationssysteme

Sommersemester 2016

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

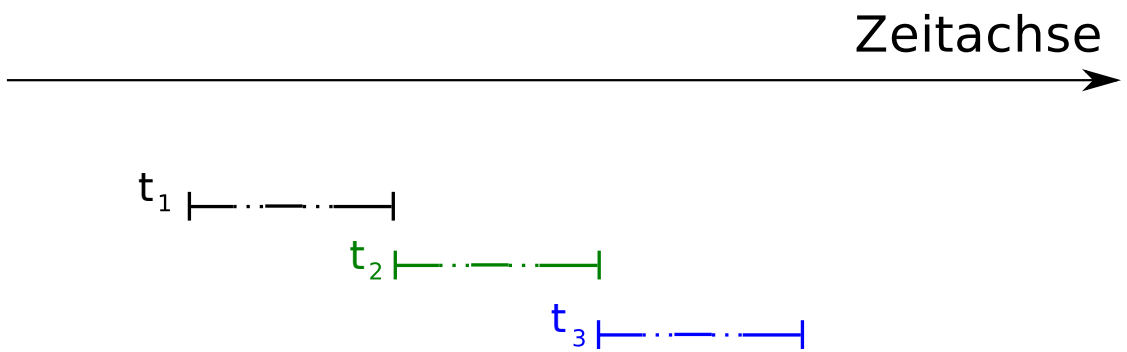
smichel@cs.uni-kl.de

Mehrbenutzersynchronisation

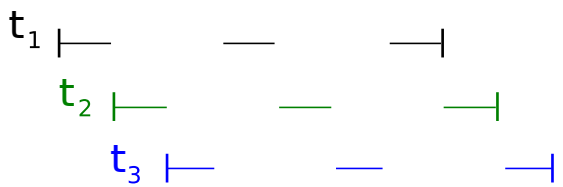
Das "**I**" in AC**I**D.

Ausführung der drei Transaktionen t_1, t_2 und t_3:

(a) im Einzelbetrieb



(b) im (verzahnten) Mehrbenutzerbetrieb



Man möchte, dass die verzahnte Ausführung "semantisch" äquivalent zu einer seriellen Ausführung (d.h. z.B. $t_1\ t_3\ t_2$) ist.

Theorie der Serialisierbarkeit: Transaktion

"Formale" Definition einer Transaktion

Operationen einer Transaktion t_i:

$r_i(x)$	$=$ Lesen des Datenobjekts x
$w_i(x)$	$=$ Schreiben des Datenobjekts x
a_i	$=$ **abort**
c_i	$=$ **commit**

Lese- und Schreiboperationen sowie mehrfache Schreiboperationen auf demselben Datenobjekt sind geordnet

Das lost-update Problem

t_1	Time	t_2
	/* $x = 100$ */	
$r(x)$	1	
	2	$r(x)$
/*update $x := x + 30$ */	3	
	4	/* update $x := x + 20$ */
$w(x)$	5	
	/* $x = 130$ */	
	6	$w(x)$
	/* $x = 120$*/	

Das lost-update Problem / Notation

- Die Essenz dieses Problems kann durch folgende Sequenz von Lese- und Schreiboperationen ausgedrückt werden:

$$r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$$

Das inconsistent-read Problem

Beispiel aus Anwendung in Bank. Aktueller Stand $x = y = 50$, also $x + y = 100$. Transaktion t_1 berechnet die Summe von x und y, während t_2 einen Wert von 10 von x nach y transferiert.

t_1	Time	t_2
	1	$r(x)$
	2	/* $x := x - 10$ */
	3	$w(x)$
/* $sum := 0$ */	4	
$r(x)$	5	
$r(y)$	6	
/* $sum := sum + x$ */	7	
/* $sum := sum + y$ */	8	
	9	$r(y)$
	10	/* $y := y + 10$ */
	11	$w(y)$

Das inconsistent-read Problem (2)

- Offensichtlich ist auch hier wieder das Problem, dass Lese- und Schreiboperationen der einzelnen Transaktionen gemischt ablaufen

$$r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)$$

Das dirty-read Problem

t_1	Time	t_2
$r(x)$	1	
/* $x := x + 100$ */	2	
$w(x)$	3	
	4	$r(x)$
	5	/* $x := x - 100$ */
failure & rollback	6	
	7	$w(x)$

Konsistenzanforderung einer Transaktion t_i

- Entweder **abort** oder **commit** aber nicht beides!
- Falls t_i ein **abort** durchführt, müssen alle anderen Operationen $p_i(x)$ vor a_i ausgeführt werden, also $p_i(x) <_i a_i$.
- Analoges gilt für das **commit**, d.h. $p_i(x) <_i c_i$ falls t_i **"committed"**.
- Wenn eine Transaktion t_i ein Datum x liest und auch schreibt, dann muss die Reihenfolge festgelegt werden, also entweder $r_i(x) <_i w_i(x)$ oder $w_i(x) <_i r_i(x)$.

Beispiel Transaktion:

$$r_1(x)\ w_1(x)\ r_1(y)\ w_1(y)\ c_1$$

Historien

- Historie enthält alle Operationen aller Transaktionen
- Historie benötigt eine Terminierungsoperation für jede TA
- Historie bewahrt alle Ordnungen innerhalb der TA
- Terminierungsoperation ist letzte Operation in jeder TA
- Konfliktoperationen sind geordnet.

Beispiel Historie:

$$H = r_1(x)\ \ r_2(x)\ \ w_1(x)\ \ r_3(x)\ \ w_3(x)\ \ w_2(y)\ \ c_3\ \ c_2\ \ w_1(y)\ \ c_1$$

Serialisierbare Historie

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

- Was bedeutet nun "äquivalent"?
- Wie entscheidet man effizient, ob eine Historie serialisierbar ist?
- Wie entscheidet man, in welcher Reihenfolge die einzelnen Operationen ausgeführt werden?

Konfliktoperationen

- Gegeben zwei Transaktionen t_i und t_j
- $r_i(x)$ und $r_j(x)$:
  - Reihenfolge der Ausführungen irrelevant, da beide Transaktionen in jedem Fall denselben Zustand lesen.
  - Operationen **stehen nicht im Konflikt** zueinander
- $r_i(x)$ und $w_j(x)$:
  - **Konflikt**, da Transaktion t_i entweder den alten **oder** den neuen Wert von x liest.
  - entweder $r_i(x)$ **vor** $w_j(x)$ oder: $w_j(x)$ **vor** $r_j(x)$ spezifizieren.
- $w_i(x)$ und $r_j(x)$: analog
- $w_i(x)$ und $w_j(x)$:
  - Reihenfolge der Ausführung entscheidend für den Zustand der Datenbasis.
  - **Konflikt**, für den die Reihenfolge festzulegen ist.

Konfliktschritte-Graph

Gegeben eine Historie H, z.B.

$$H = r_1(x)\ \ r_2(x)\ \ w_1(x)\ \ r_3(x)\ \ w_3(x)\ \ w_2(y)\ \ c_3\ \ c_2\ \ w_1(y)\ \ c_1$$

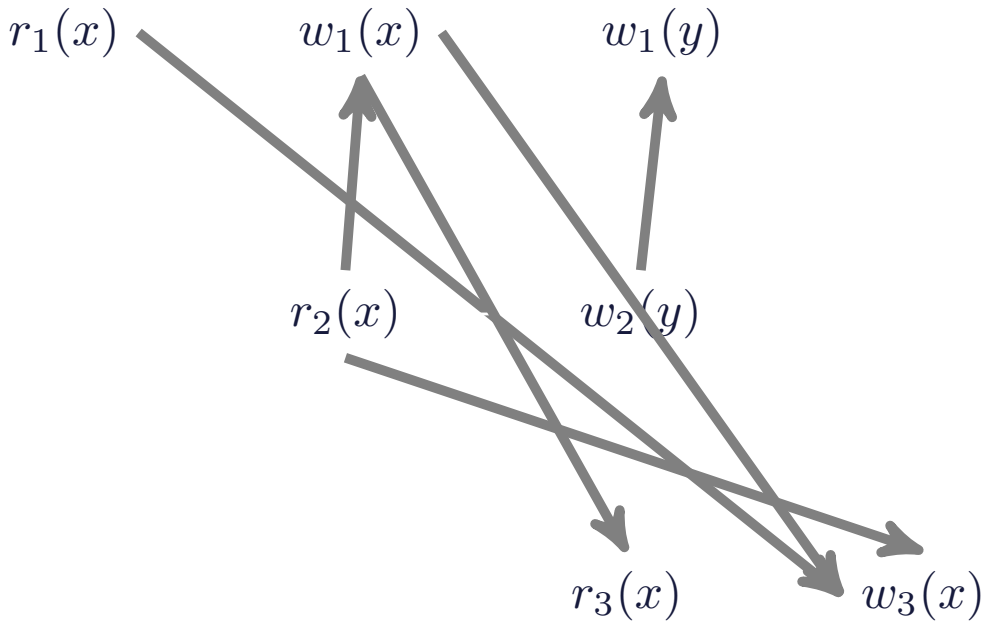
Wir definieren einen Graphen D, den sogenannten **Konfliktschritte-Graph**, wie folgt: Als Kanten haben wir alle Operationen der in der Historie beteiligten Transaktionen. D hat gerichtete Kanten zwischen den einzelnen Konfliktoperationen.
Für das Beispiel oben:

$D(H) =$

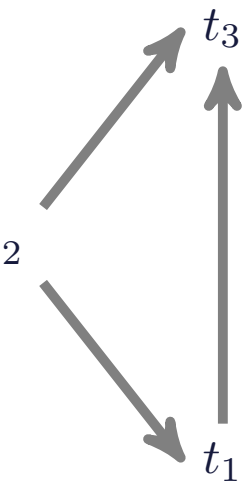
Zwei Historien s und s' heißen äquivalent, wenn Ihre Konfliktschritte-Graphen identisch sind.

Serialisierbarkeitsgraph

$D =$



$SG =$



- **Konfliktschritte-Graph D beschreibt einzelne Konflikte zwischen Operationen $p_i(x_i)$**
- Kante $w_1(x) \rightarrow r_3(x)$ aus D führt zur Kante $t_1 \rightarrow t_3$ des **Serialisierbarkeitsgraphen** (aka. Konfliktgraph) SG
- Weitere Kanten analog
- **Ist die zugrunde liegende Historie serialisierbar?**

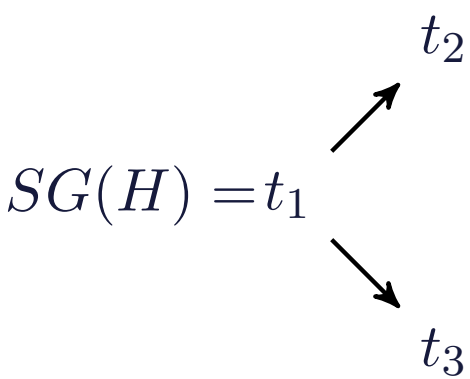
Serialisierbarkeitstheorem

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

Historie

$$H = w_1(x)\ w_1(y)\ c_1)\ r_2(x)\ r_3(y)\ w_2(x)\ c_2\ w_3(y)\ c_3$$

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 Recovery

Wir sagen, dass in der Historie H Transaktion t_i von t_j liest, wenn folgendes gilt:

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

Intuition: t_i liest ein Datum in genau dem Zustand, den t_j geschrieben hat.

Rücksetzbare Historien (RC)

Eine Historie heißt zurücksetzbar (recoverable, RC), 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.
- Ansonsten ist diese TA womöglich nicht rücksetzbar, da sie mit einem Wert gerechnet hat, der nie existiert hat
- Nicht rücksetzbar $\Rightarrow$ verletzt ACID
- Warum?

Merkregel: TA darf erst committen, wenn alle TAs, von denen sie gelesen hat, beendet sind.

Historien 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.	$a_1(abort)$				

Warum ist das hier rücksetzbar?

Historien ohne kaskadierendes Rücksetzen (ACA)

Historien ohne kaskadierendes Rücksetzen (avoids cascading aborts: ACA): Eine Historie H vermeidet kaskadierendes Rücksetzen, wenn für je zwei TAs t_i und t_j gilt: $c_j <_H r_i(x)$ gilt, wann immer t_i ein Datum x von t_j liest.

Merkregel: Es dürfen nur Änderungen von abgeschlossenen TAs gelesen werden.

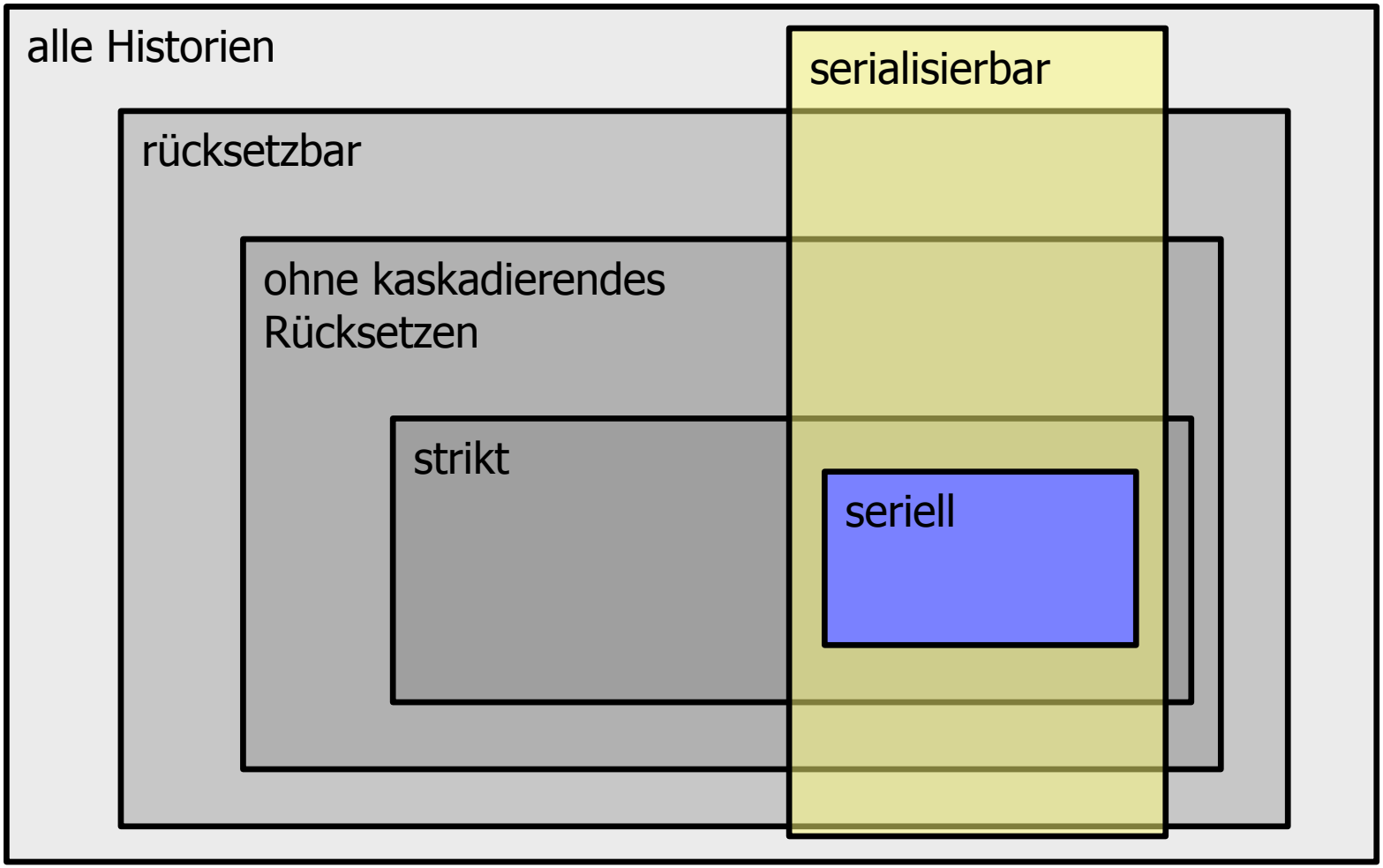
Strikte Historien

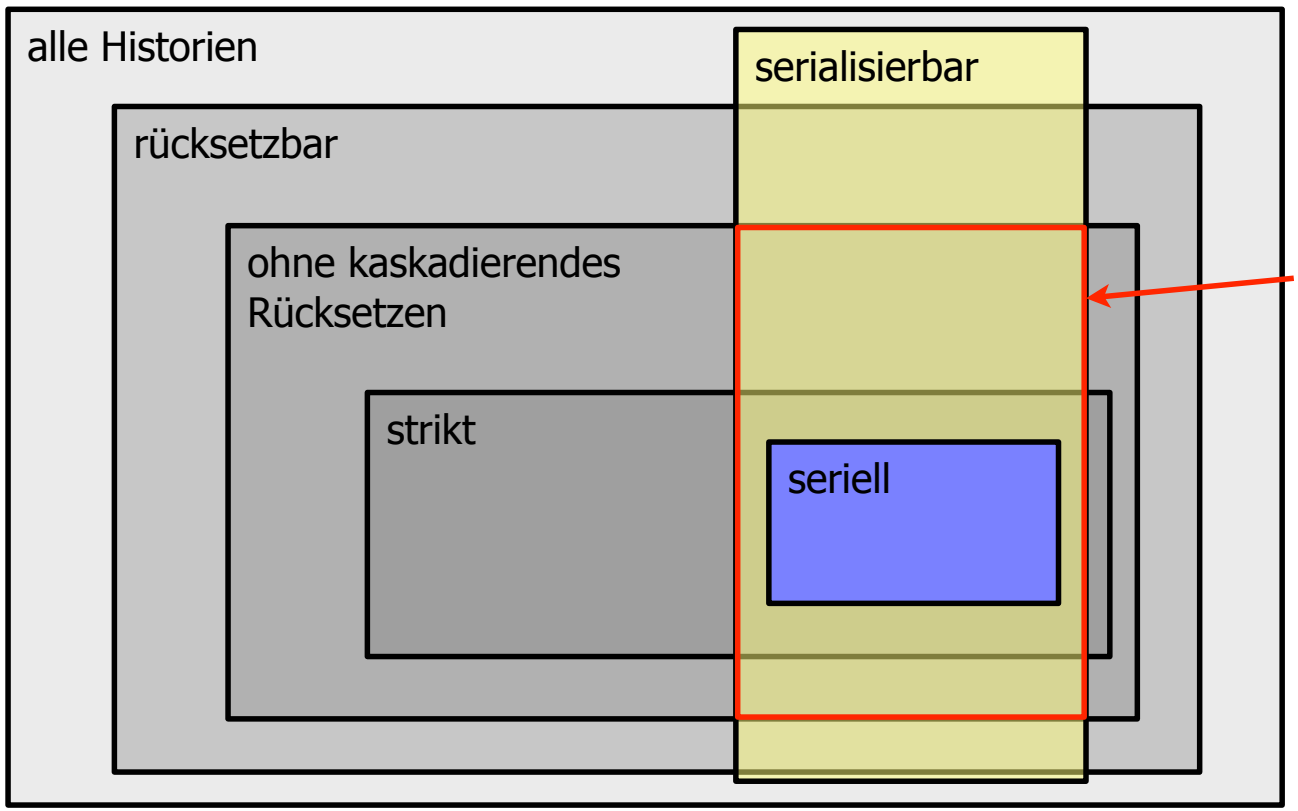
Eine Historie H heißt strikt, wenn für je zwei Transaktionen t_i und t_j gilt:

Wenn $w_j(A) <_H o_i(A)$ mit $o_i = r_i$ oder $o_i = w_i$, dann muss gelten:

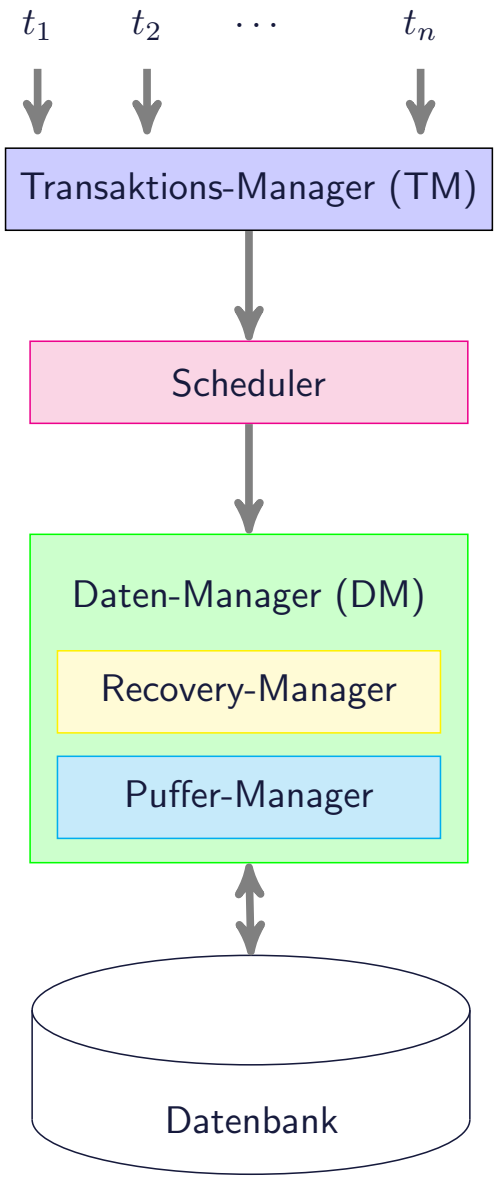
- $a_j <_H o_i(A)$ oder
- $c_j <_H o_i(A)$

Merkregel: Geänderte Daten nicht-abgeschlossener TAs dürfen weder gelesen noch überschrieben werden.


alle Historien
serialisierbar
rücksetzbar
ohne kaskadierendes
Rücksetzen
strikt
seriell


alle Historien
serialisierbar
rücksetzbar
ohne kaskadierendes Rücksetzen
strikt
seriell
DBMS erzeugt Historie aus dieser Schnittmenge

Der Datenbank-Scheduler



- TM: Informationen über TA, welche Schritte als nächstes ausgeführt werden können.
- Scheduler: Bekommt Schedules vom TM und muss diese in einen serialisierbaren Schedule umwandeln.
- Welcher verarbeitet wird von Daten-Manager (DM)

Sperrprotokolle - Allgemeines

Allgemeine Idee

- **Zugriff auf gemeinsam genutzte Daten wird durch Sperren synchronisiert**
- Hier: Ausschließlich konzeptionelle Sichtweise und gleichförmige Granulate wie Seiten (keine Implementierungstechnik, keine multiplen Granulate usw.)

Allgemeine Vorgehensweise

- Scheduler fordert für die betreffende TA für jeden ihrer Schritte eine Sperre an
- Jede Sperre wird in einem spezifischen Modus angefordert (read oder write)
- Falls das Datenelement noch nicht in einem unverträglichen Modus gesperrt ist, wird die Sperre gewährt; sonst ergibt sich ein Sperrkonflikt und die TA wird blockiert, bis die Sperre freigegeben wird.

Sperrbasierte Synchronisation

Zwei Sperrmodi:

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

Verträglichkeitsmatrix (auch Kompatibilitätsmatrix genannt) und neuer Modus:

aktueller Modus des Objekts x

angeforderte Sperre	NL	R	X
$rl(x)$	+	+	-
$wl(x)$	+	-	-

neuer Modus des Objekts x

	NL	R	X
$rl(x)$	R	R	-
$wl(x)$	X	-	-

Zwei-Phasen Sperrprotokoll (2PL): Definition

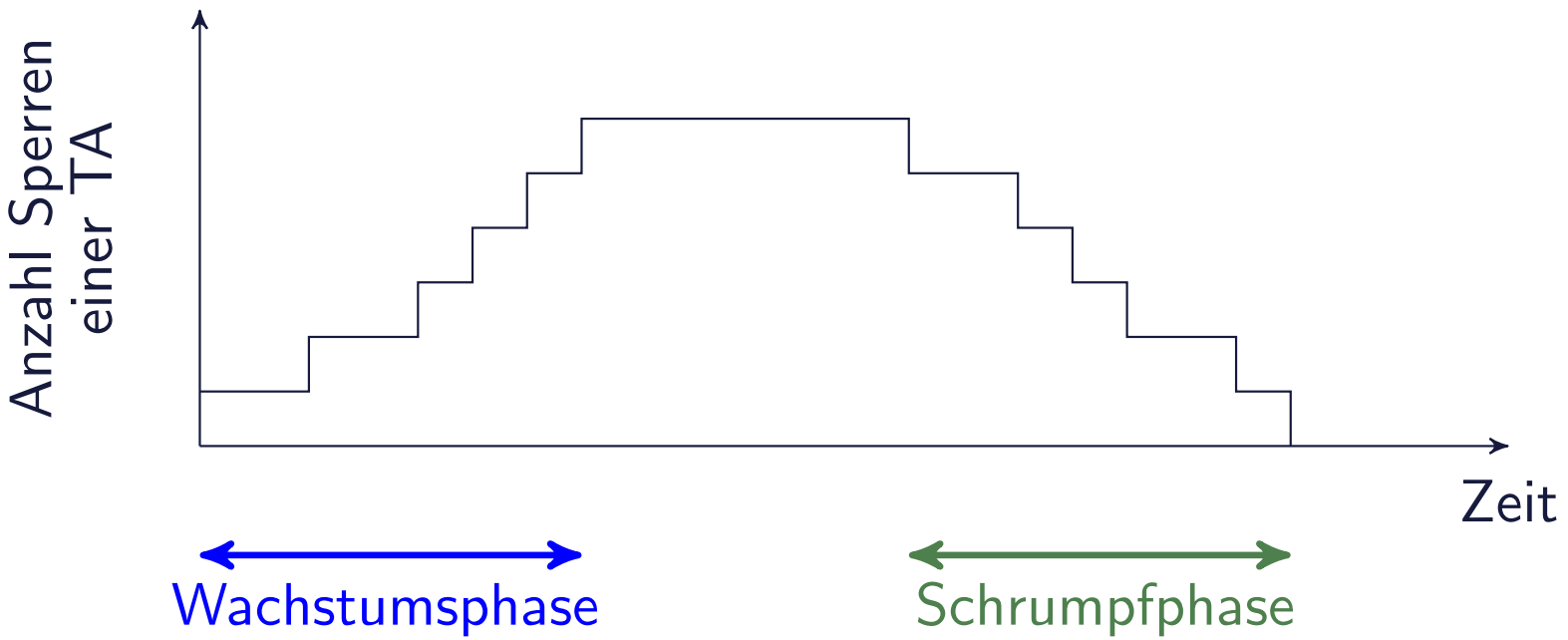
1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muss vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon benutzt, nicht erneut an.
3. Eine Transaktion muss die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereit - 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 (2PL): Grafik

Definition: 2PL

Ein Sperrprotokoll ist **zweiphasig** (2PL), wenn für jede TA t_i kein Lock (d.h. ql_i Schritt) dem ersten Unlock (d.h. ou_i Schritt) folgt ($o, q \in \{r,w\}$).

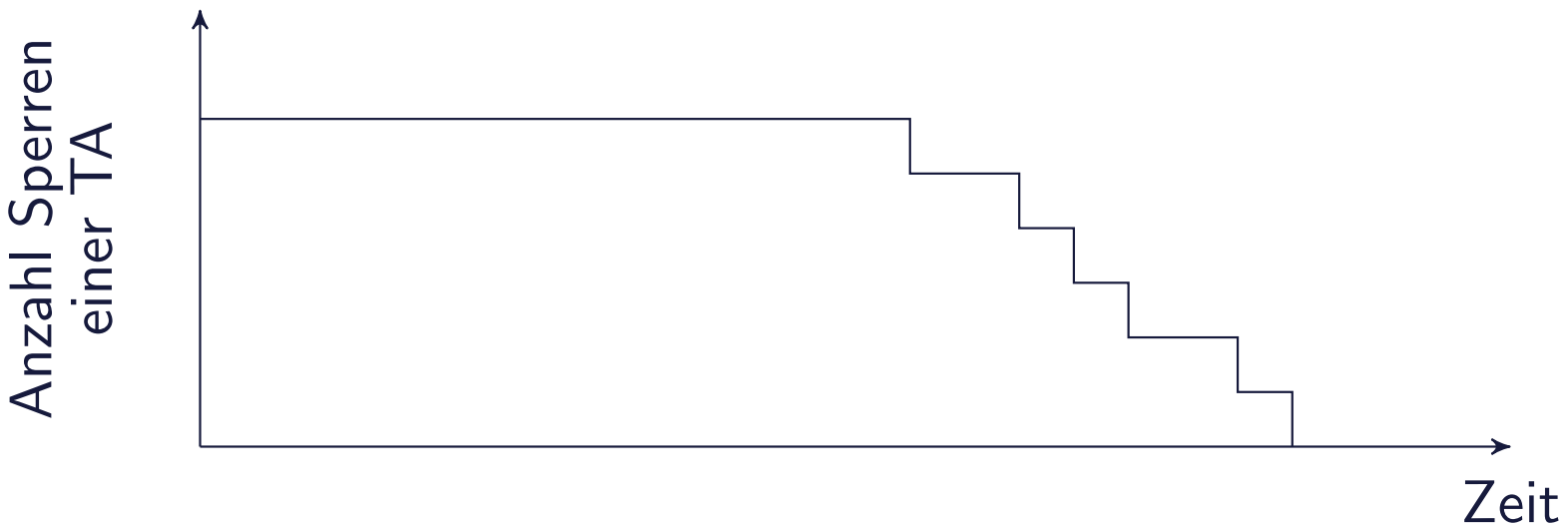
2PL erzeugt nur serialisierbare Historien $\Rightarrow$ 2PL garantiert Serialisierbarkeit



Konservatives 2PL (C2PL)

Definition: Konservatives 2PL

Unter konservativem 2PL (C2PL) fordert jede TA alle Sperren an, bevor sie den erste Read- oder Write-Schritt ausführt (**Preclaiming**).
In der Praxis nur eingeschränkt anwendbar, da alle Sperren schon zu Beginn der TA bekannt sein müssen.

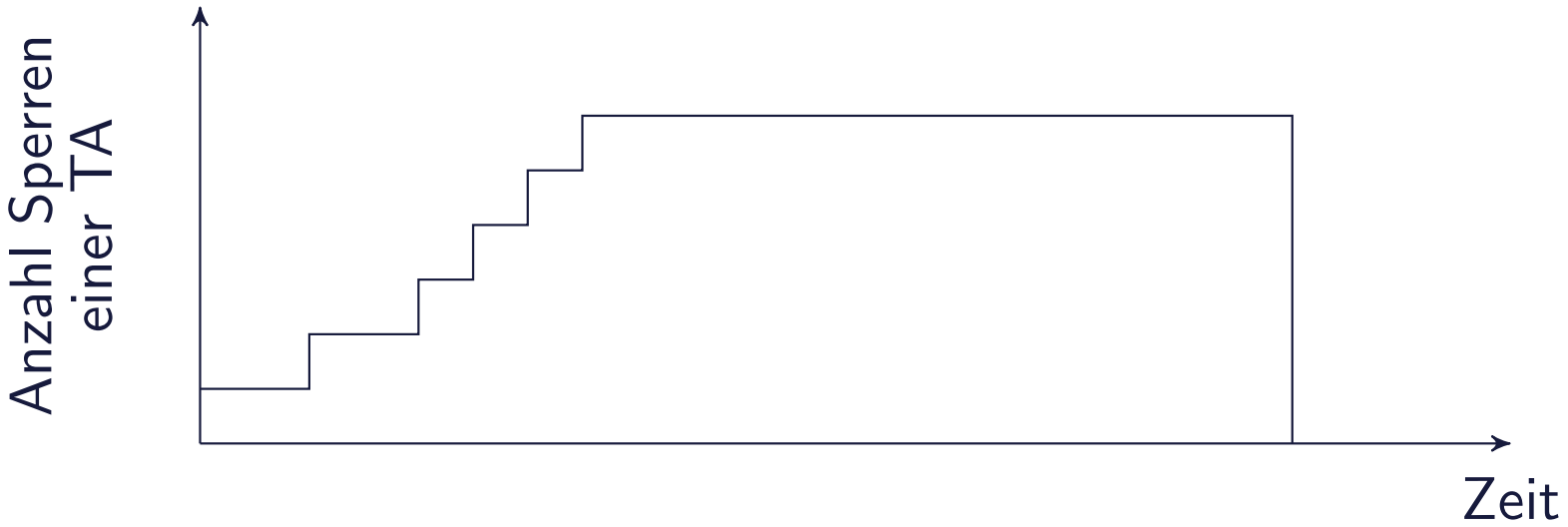


Striktes 2PL (S2PL) und Starkes 2PL (SS2PL)

Definition: Striktes 2PL

Unter striktem 2PL (S2PL) werden **alle exklusiven Sperren** (wl) einer TA bis zur ihrer Terminierung gehalten.

Wird in praktischen Implementierungen am häufigsten eingesetzt. Erzeugt **serialisierbare und strikte Historien**



Definition: Starkes 2PL

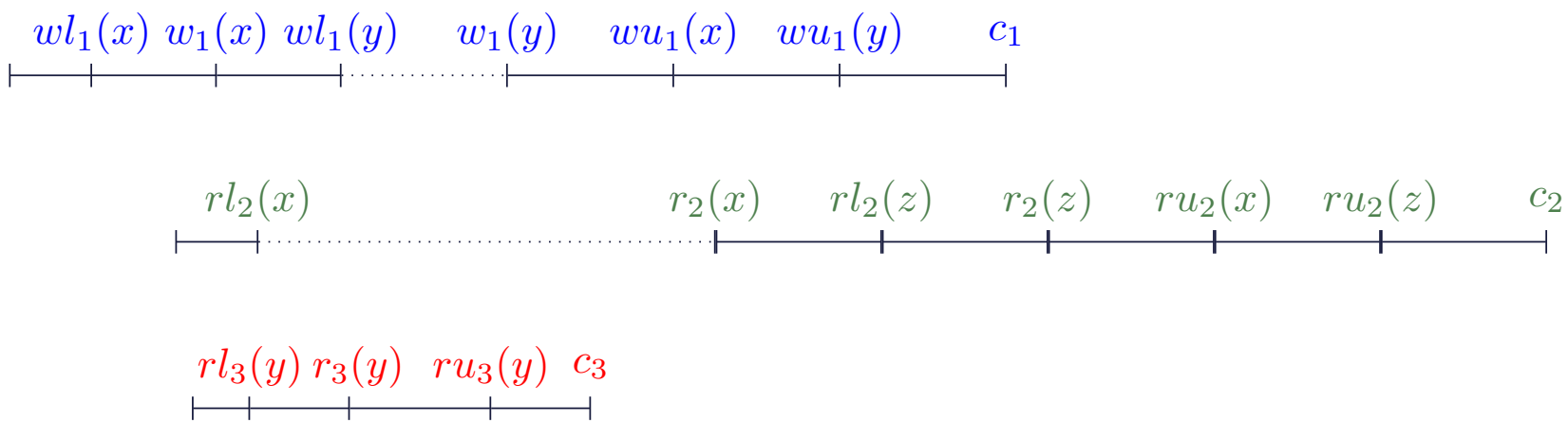
Unter starkem 2PL (strong strict 2PL, SS2PL) werden **alle Sperren** (wl, rl) einer TA bis zur ihrer Terminierung gehalten.

Sperrprotokoll 2PL

Beispiel: Eingabe-Schedule

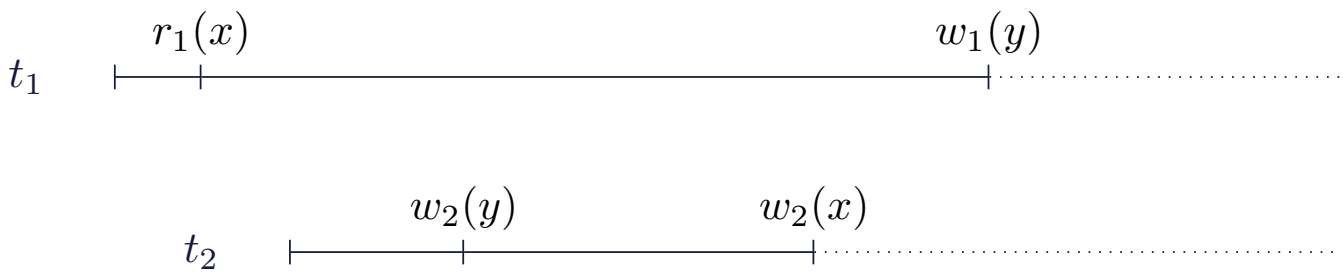
$$s_1 = w_1(x)\ r_2(x)\ r_3(y)\ r_2(z)\ w_1(y)\ c_3\ c_1\ c_2$$

2PL Scheduler transformiert s_1 z.B. in folgende Ausgabe-Historie:



Deadlocks

- **... werden verursacht durch zyklisches Warten auf Sperren**
- Beispiel



Deadlock-Erkennung

Aufbau eines dynamischen **Wait-for-Graph** (WfG) mit aktiven TAs als Knoten und Wartebeziehungen als Kanten: Eine Kante von t_i nach t_j ergibt sich, wenn t_i auf eine von t_j gehaltene Sperre wartet.

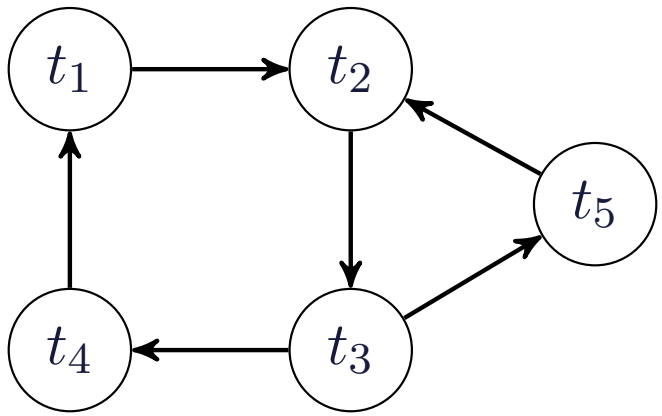
Testen des WfG zur Zyklenerkennung

- kontinuierlich (bei jedem Blockieren)
- periodisch (z.B. einmal pro Sekunde)

Erkennung von Verklemmung

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.
- Verschiedene Strategien welche TA zurückgesetzt werden soll (jüngste, älteste, nach # Sperren etc.)

Transaktionsverwaltung in SQL92

Problem:

Serialisierbarkeit kann zu Performanzproblemen führen.

Idee:

Isolationsgarantien abschwächen, um höhere Performanz zu erreichen.

Vier sogenannte Isolationslevel:

set transaction
[read only, | read write,]
[**isolation level**
read uncommitted, |
read committed, |
repeatable read, |
serializable,] |
[**diagnostics size ...,**]

Isolationsstufen

read uncommitted:

- Schwächste Konsistenzstufe
- Liest möglicherweise inkonsistente/uncommitted data
- Nur für read-only TAs
- Geeignet für "browsing"
- Keine locks für read-Operationen

read committed:

- Nur committed Werte dürfen gelesen werden
- Auch für Schreib-TAs erlaubt
- Locks zum Schreiben im strikten 2PL
- Locks zum Lesen nur kurzzeitig gehalten
- Non-repeatable read möglich
- Phantom-Problem möglich

Isolationsstufen (2)

repeatable read:

- Locks zum Schreiben mit striktem 2PL
- Locks zum Lesen mit striktem 2PL
- Non-repeatable read ausgeschlossen
- Phantom-Problem möglich

serializable:

- Volle Serialisierbarkeit
- Nicht immer default
- Locks zum Schreiben mit striktem 2PL
- Locks zum Lesen mit striktem 2PL
- Zusätzlich wird der Zugriff über Indexe/Prädikate gelockt

Probleme mit Locking

- Transaktionen halten locks länger als zum Lesen/Schreiben eigentlich notwendig (wegen 2PL)
- Deadlock-Erkennung aufwendig
- Pessimistischer Ansatz: versteckte Annahme, dass es schief gehen wird (Murphy's Law)
- Was, wenn es hauptsächlich Lesetransaktionen gibt?
- Dann blockiert jedes Write eine große Zahl von Lesetransaktionen
- $\Rightarrow$ Lesetransaktionen müssen warten
- $\Rightarrow$ Schlechte Performance

Mehr dazu in der VL Datenbanksysteme im Winter.