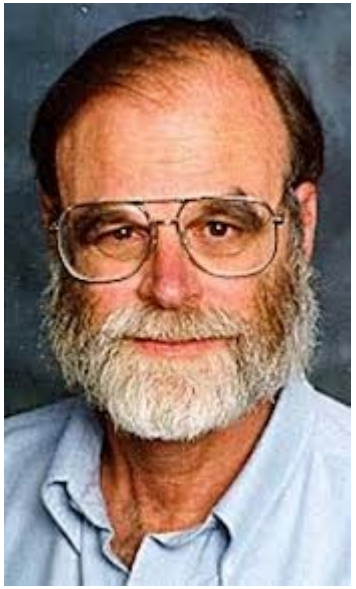
Transaktionsverarbeitung und Nebenläufigkeitskontrolle



Jim Gray (1944-2007)

Transaktionsverwaltung

1. Lese den Kontostand von *A* in die Variable *a*: **read**(*A*,*a*);
2. Reduziere den Kontostand um 50.- €: *a*:= *a* – 50;
3. Schreibe den neuen Kontostand in die Datenbasis: **write**(*A*,*a*);
4. Lese den Kontostand von *B* in die Variable *b*: **read**(*B*,*b*);
5. Erhöhe den Kontostand um 50,- €: *b* := *b* + 50;
6. Schreibe den neuen Kontostand in die Datenbasis: **write**(*B*, *b*);

Operationen auf Transaktions-Ebene

- **begin of transaction (BOT)**: Mit diesem Befehl wird der Beginn einer eine Transaktion darstellende Befehlsfolge gekennzeichnet.
- **commit**: Hierdurch wird die Beendigung der Transaktion eingeleitet. Alle Änderungen der Datenbasis werden durch diesen Befehl festgeschrieben, d.h. sie werden dauerhaft in die Datenbank eingebaut.
- **abort**: Dieser Befehl führt zu einem Selbstabbruch der Transaktion. Das Datenbanksystem muss sicherstellen, dass die Datenbasis wieder in den Zustand zurückgesetzt wird, der vor Beginn der Transaktionsausführung existierte.

Abschluss einer Transaktion

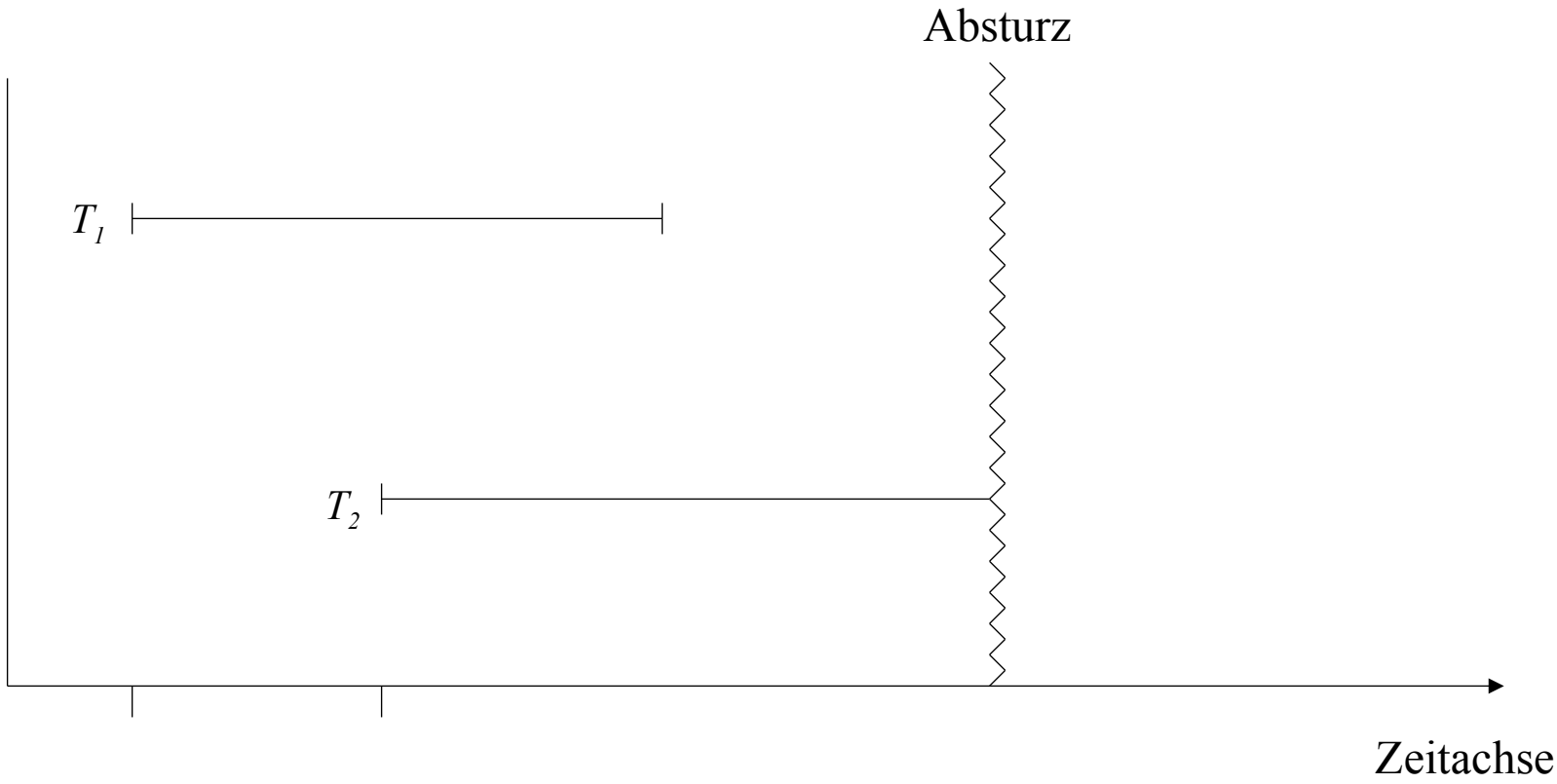
Für den Abschluss einer Transaktion gibt es zwei Möglichkeiten:

1. Den erfolgreichen Abschluss durch ein **commit**.
2. Den erfolglosen Abschluss durch ein **abort**.

Eigenschaften von Transaktionen:
ACID

- **A**tomicity
  - Alles oder nichts: Die Transaktion wird komplett ausgeführt oder bleibt wirkungslos.
- **C**onsistency
  - Konsistenter Zustand der DB → konsistenter Zustand
- **I**solation
  - Jede Transaktion hat die DB „für sich allein“, keine Beeinflussung durch parallel laufende Transaktionen
- **D**urability (Dauerhaftigkeit)
  - Änderungen erfolgreicher Transaktionen dürfen nie verloren gehen

Eigenschaften von Transaktionen



Transaktionsbeginn und –ende relativ zu einem Systemabsturz

- Änderungen von T_1 bleiben erhalten
- Änderungen von T_2 werden rückgängig gemacht

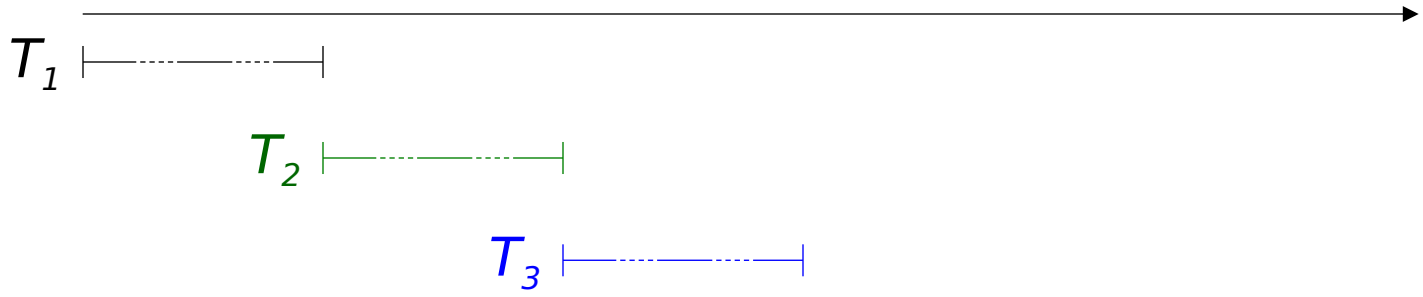
Transaktionsverwaltung in SQL

- **commit work**: Die in der Transaktion vollzogenen Änderungen werden – falls keine Konsistenzverletzung oder andere Probleme aufgedeckt werden – festgeschrieben.
  Das Schlüsselwort **work** ist optional, d.h. das Transaktionsende kann auch einfach mit **commit** „*befohlen*“ werden.
- **rollback work**: Alle Änderungen sollen zurückgesetzt werden.
  Anders als der **commit**-Befehl muss das DBMS die „*erfolgreiche*“ Ausführung eines rollback-Befehls immer garantieren können.

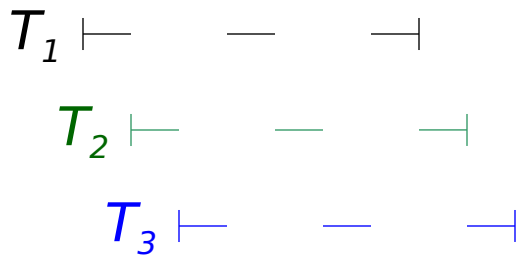
Mehrbenutzersynchronisation

Ausführung der drei Transaktionen T_1, T_2 und T_3:

(a) im Einzelbetrieb und



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



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)

Fehler bei unkontrolliertem Mehrbenutzerbetrieb I

Verlorengegangene Änderungen (*lost update*)

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

Fehler bei unkontrolliertem Mehrbenutzerbetrieb II

Abhängigkeit von nicht freigegebenen Änderungen

(dirty read)

Schritt	T_1	T_2
1.	read(A,a_1)	
2.	$a_1 := a_1 - 300$	
3.	write(A,a_1)	
4.		read(A,a_2)
5.		$a_2 := a_2 * 1.03$
6.		write(A,a_2)
7.	read(B,b_1)	
8.	. . .	
9.	abort	

Fehler bei unkontrolliertem Mehrbenutzerbetrieb III

Überschreiben von Daten, die noch gelesen werden
(*unrepeatable read*)

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

Fehler bei unkontrolliertem Mehrbenutzerbetrieb IV

Phantomproblem

T_1	T_2
	select sum(KontoStand)
	from Konten
insert into Konten 	
values (*C*,1000,...)	
	select sum(Kontostand)
	from Konten

Serialisierbarkeit

- Historie ist „äquivalent“ zu einer seriellen Historie
- dennoch parallele (verzahnte) Ausführung möglich

Serialisierbare Historie von T_1 und T_2?

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

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

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

Nicht serialisierbare Historie

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

Anmerkungen

- Aus Sicht des DBMS ist diese Historie nicht serialisierbar (sowohl T1 | T2 als auch T2 | T1 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 verzahnte Überweisungs-Transaktionen

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 - 100$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 + 100$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

Eine Überweisung (T_1) und eine Zinsgutschrift (T_3)

Schritt	T_1	T_3
1.	**BOT**	
2.	read(A,a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A,a_1)	
5.		**BOT**
6.		read(A,a_2)
7.		$a_2 := a_2 * 1.03$
8.		write(A,a_2)
9.		read(B,b_2)
10.		$b_2 := b_2 * 1.03$
11.		write(B,b_2)
12.		**commit**
13.	read(B,b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B,b_1)	
16.	**commit**	

Theorie der Serialisierbarkeit

„Formale“ Definition einer Transaktion

Operationen einer Transaktion T_i

- $r_i(A)$ zum Lesen des Datenobjekts A,
- $w_i(A)$ zum Schreiben des Datenobjekts A,
- a_i zur Durchführung eines **aborts**,
- c_i zur Durchführung des **commit**.

Theorie der Serialisierbarkeit

Konsistenzanforderung einer Transaktion T_i

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

Theorie der Serialisierbarkeit II

Reihenfolge von Operationen

- $r_i(A)$ und $r_j(A)$: In diesem Fall ist die Reihenfolge der Ausführungen irrelevant, da beide TAs in jedem Fall denselben Zustand lesen. Diese beiden Operationen stehen also nicht in Konflikt zueinander, so dass in der Historie ihre Reihenfolge zueinander irrelevant ist.

- $r_i(A)$ und $w_j(A)$: Hierbei handelt es sich um einen Konflikt, da T_i entweder den alten oder den neuen Wert von A liest. Es muss also entweder $r_i(A)$ *vor* $w_j(A)$ oder $w_j(A)$ *vor* $r_i(A)$ spezifiziert werden.

- $w_i(A)$ und $r_j(A)$: analog

- $w_i(A)$ und $w_j(A)$: Auch in diesem Fall ist die Reihenfolge der Ausführung entscheidend für den Zustand der Datenbasis; also handelt es sich um Konfliktoperationen, für die die Reihenfolge festzulegen ist.

Formale Definition einer Historie

Eine Historie *H* ist charakterisiert durch

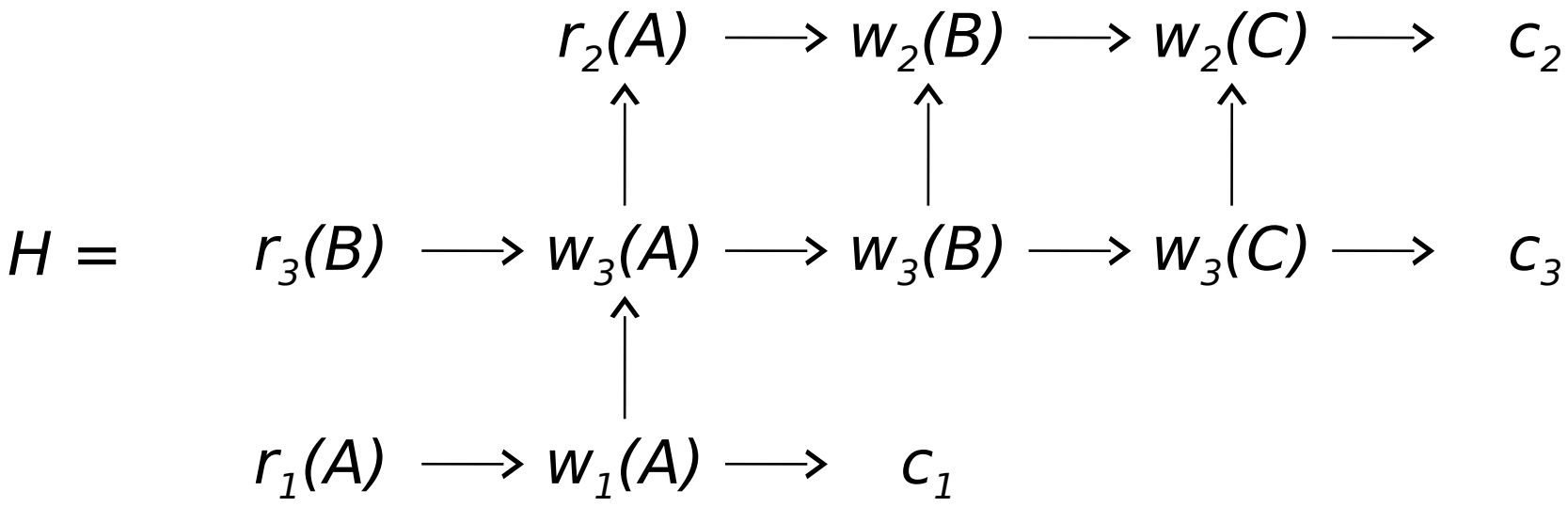
- Menge von Transaktionen $\{T_1, T_2, \ldots, T_n\}$, $H = \bigcup_{i=1}^{n} T_i$
- (partielle) Ordnung $<_H$ zwischen den Elementaroperationen dieser Transaktionen

Konsistenzanforderungen:

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

Historie für drei Transaktionen

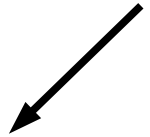
Beispiel-Historie für 3 TAs



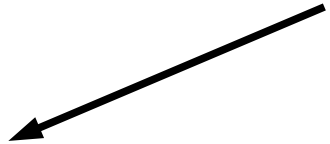
Äquivalenz zweier Historien

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

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



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



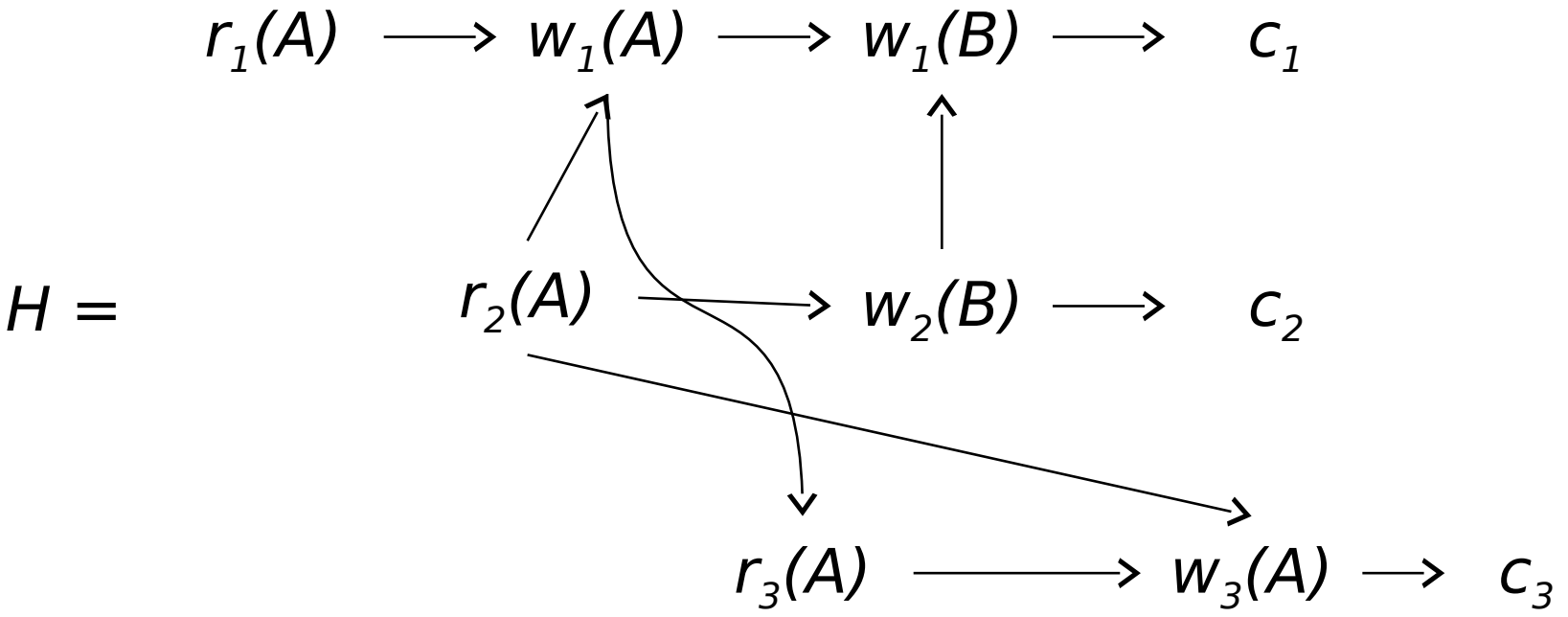
$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

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

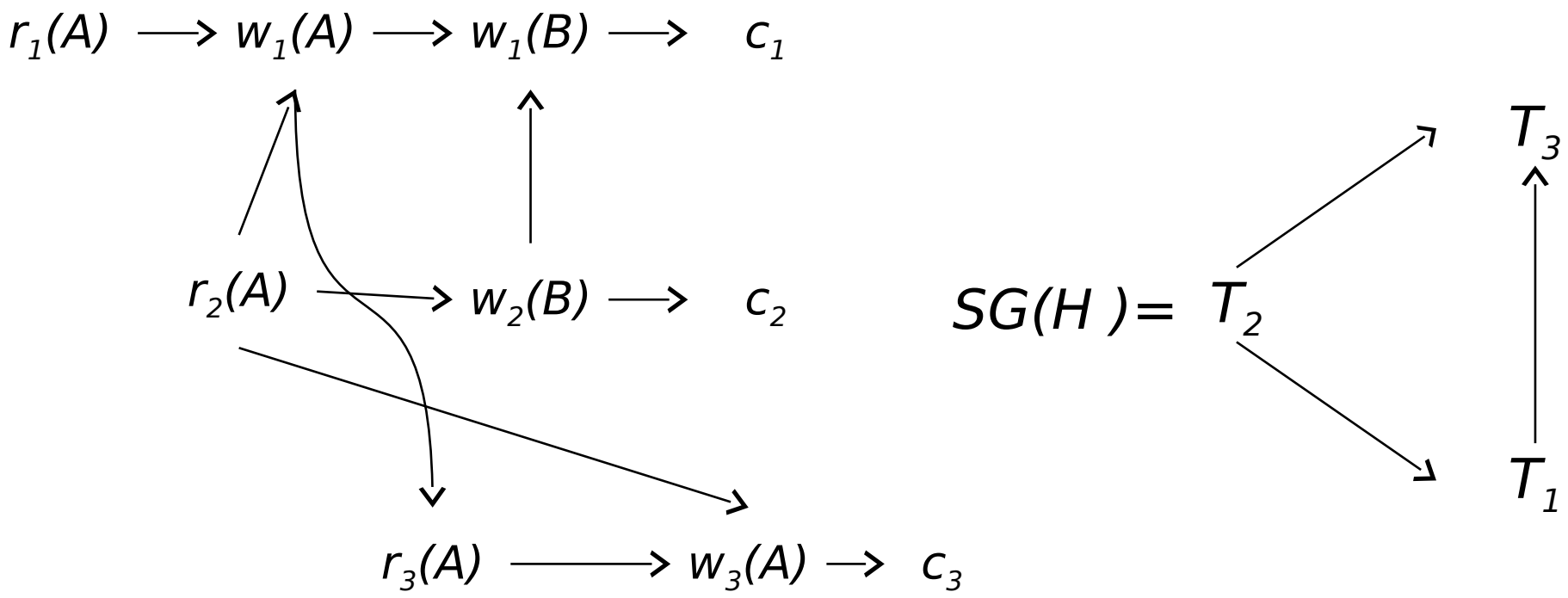
Serialisierbare Historie

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

Historie und zugehöriger Serialisierbarkeitsgraph



Serialisierbarkeitsgraph



- $w_1(A) \rightarrow r_3(A)$ der Historie H führt zur Kante $T_1 \rightarrow T_3$ des SG
- weitere Kanten analog
- „Verdichtung“ der Historie

Topolog. Ordnung: $T_2|T_1|T_3$

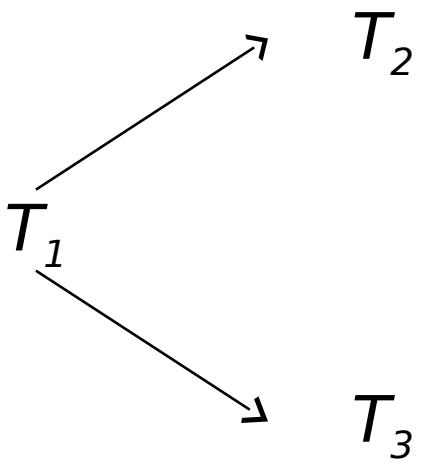
Serialisierbarkeitstheorem

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

Historie

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

Serialisierbarkeitsgraph



Topologische Ordnung(en)

$$H_s^1 = T_1 | T_2 | T_3$$

$$H_s^2 = T_1 | T_3 | T_2$$

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

Eigenschaften von Historien bezüglich der Recovery

Terminologie

Wir sagen, dass in der Historie H Transaktion T_i von T_j liest, wenn folgendes gilt:

- T_j schreibt mindestens ein Datum A, das T_i nachfolgend liest, also:

$$w_j(A) <_H r_i(A)$$

- T_j wird (zumindest) nicht vor dem Lesevorgang von T_i zurückgesetzt, also:

$$a_j \not< r_i(A)$$

- Alle anderen zwischenzeitlichen Schreibvorgänge auf A durch andere Transaktionen T_k werden vor dem Lesen durch T_i zurückgesetzt. Falls also ein $w_k(A)$ mit $w_j(A) < w_k(A) < r_i(A)$ existiert, so muss es auch ein $a_k < r_i(A)$ *geben.*

Eigenschaften von Historien bezüglich der Recovery

Rücksetzbare Historien

Eine Historie heißt rücksetzbar, falls immer die schreibende Transaktion (in unserer Notation T_j) vor der lesenden Transaktion (T_i genannt) ihr **commit** durchführt,

also: $c_j <_H c_i$.

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

Eigenschaften von Historien bezüglich der Recovery

Beispiel-Historie mit kaskadierendem Rücksetzen

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

Historien ohne kaskadierendes Rücksetzen

Eine Historie vermeidet kaskadierendes Rücksetzen, wenn für je zwei TAs T_i und T_j gilt:

- $c_j <_H r_i(A)$ gilt, wann immer T_i ein Datum A von T_j liest.

Strikte Historien

Strikte Historien vermeiden das Rücksetzen von Transaktionen

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

$$w_j(A) <_H o_i(A)$$

Dann muss gelten:

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

Rücksetzbarkeit von Historien

T_i liest von T_j

Tj	Ti
wj(A)	
	ri(A)
	ci
aj	

nicht rück-setzbare Historie

Tj	Ti	Tk
wj(A)		
		wk(A)
		ak
	ri(A)	
	ci	
cj		

rücksetzbare Historie

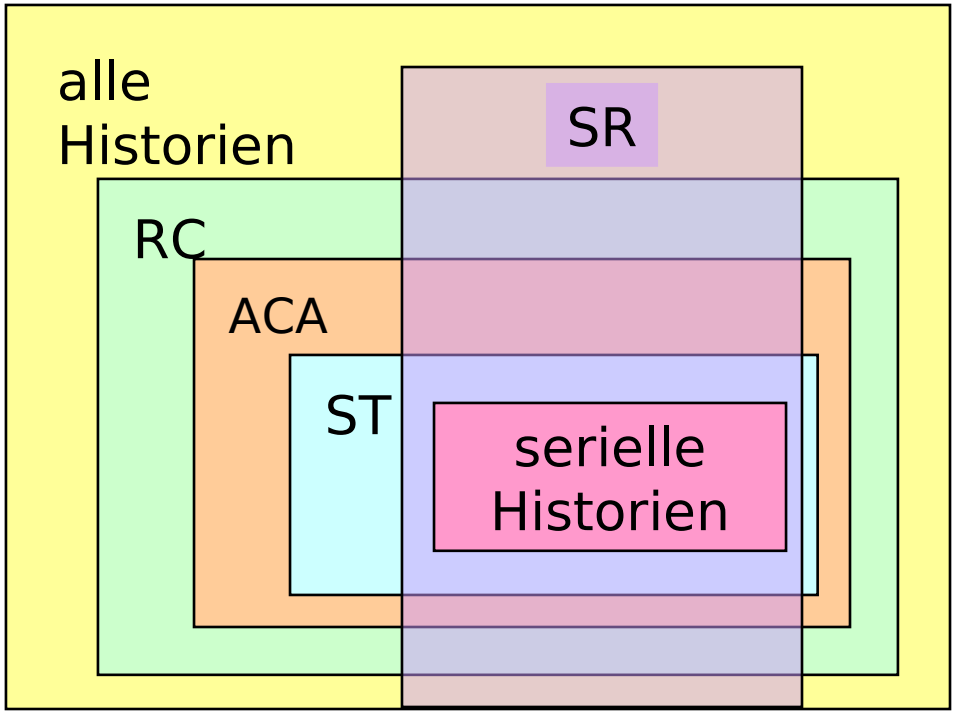
Tj	Ti
wj(A)	
	ri(A)
cj	
	wi(A)

- rücksetzbare H.
- ohne kaskad. Rücksetzen
- strikt

Tj	Ti
wj(A)	
	wi(A)
	wi(B)
	ci
aj	

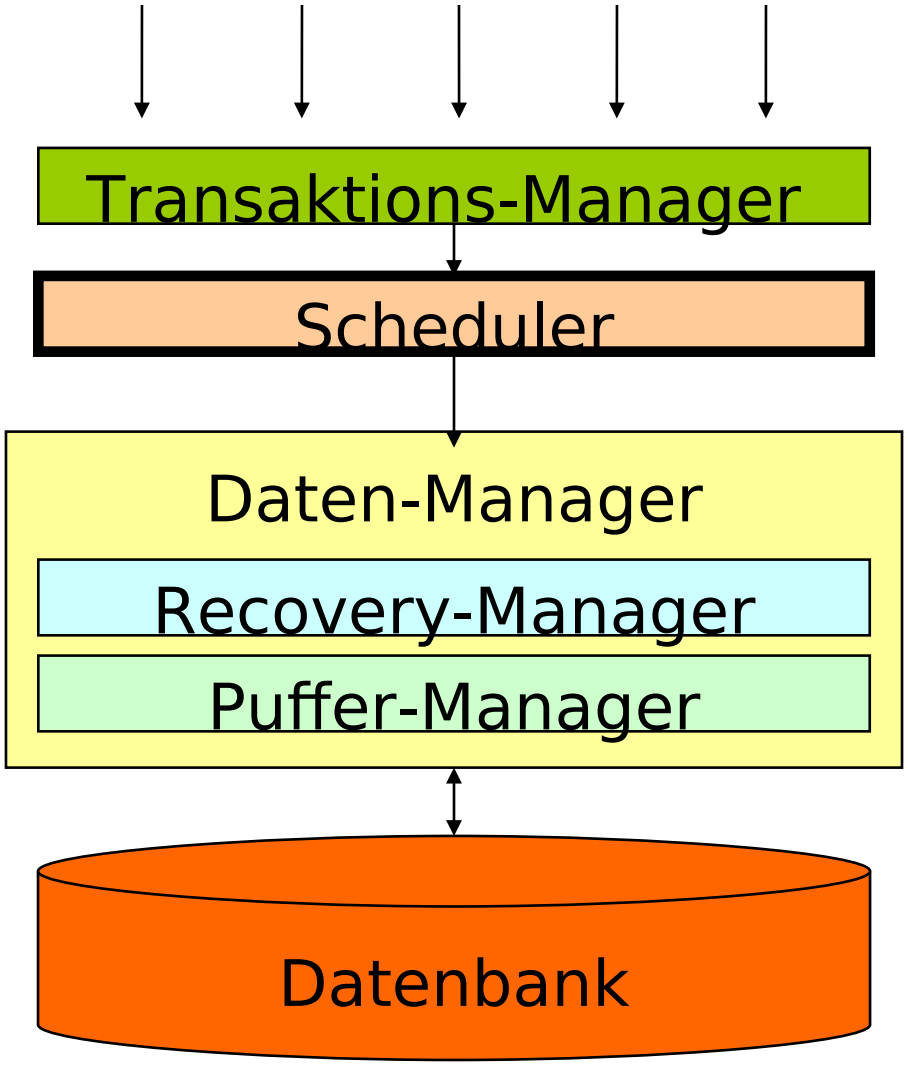
- rücksetzbare H.
- ohne kaskad. Rücksetzen
- nicht strikt

Beziehungen zwischen den Klassen von Historien



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

Der Datenbank-Scheduler



Gewährleistung von Serialisierbarkeit und Rücksetzbarkeit durch den Scheduler

- im Wesentlichen durch
  - Verzögern von Operationen
  - Abbruch von Transaktionen
- Mögliche Arten von Synchronisierung:
  - (serielle Historien)
  - (Konstruktion von Serialisierbarkeits-Graphen)
  - sperrbasierte Synchronisation
  - Synchronisation mit Zeitstempeln
  - optimistische Synchronisation

Sperrbasierte Synchronisation

Zwei Sperrmodi

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

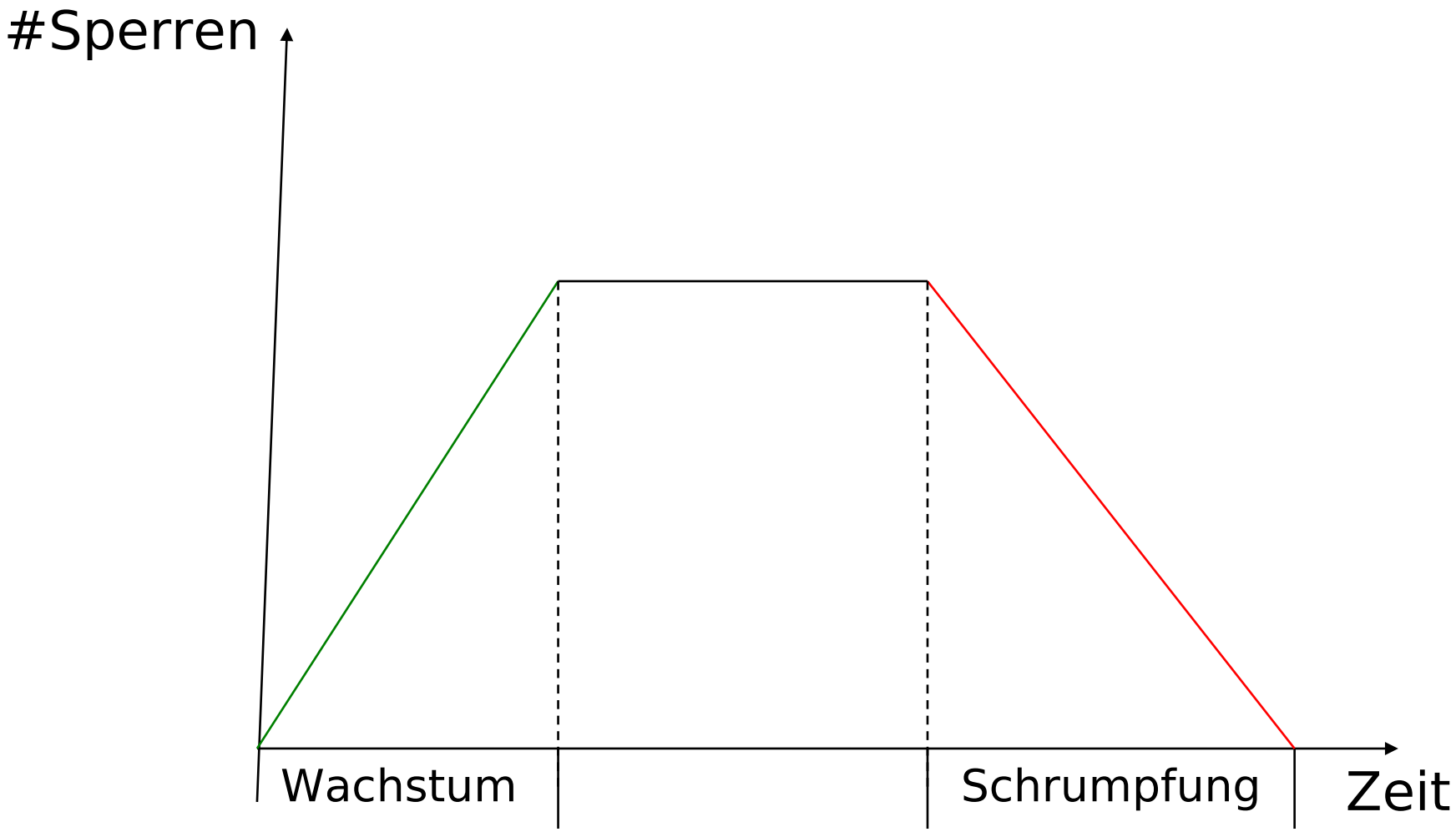
- *Verträglichkeitsmatrix* (auch *Kompatibilitätsmatrix* genannt)

	NL	*S*	*X*
S	✓	✓	-
X	✓	-	-

Zwei-Phasen-Sperrprotokoll: Definition

1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muss vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. eine Transaktion muss die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht – bis die Sperre gewährt werden kann.
4. Jede Transaktion durchläuft zwei Phasen:
   - Eine *Wachstumsphase*, in der sie Sperren anfordern, aber keine freigeben darf und
   - eine *Schrumpfphase*, in der sie ihre bisher erworbenen Sperren freigibt, aber keine weiteren anfordern darf.
9. Bei EOT (Transaktionsende) muss eine Transaktion alle ihre Sperren zurückgeben.

Zwei-Phasen Sperrprotokoll: Grafik



Verzahnung zweier TAs gemäß 2PL

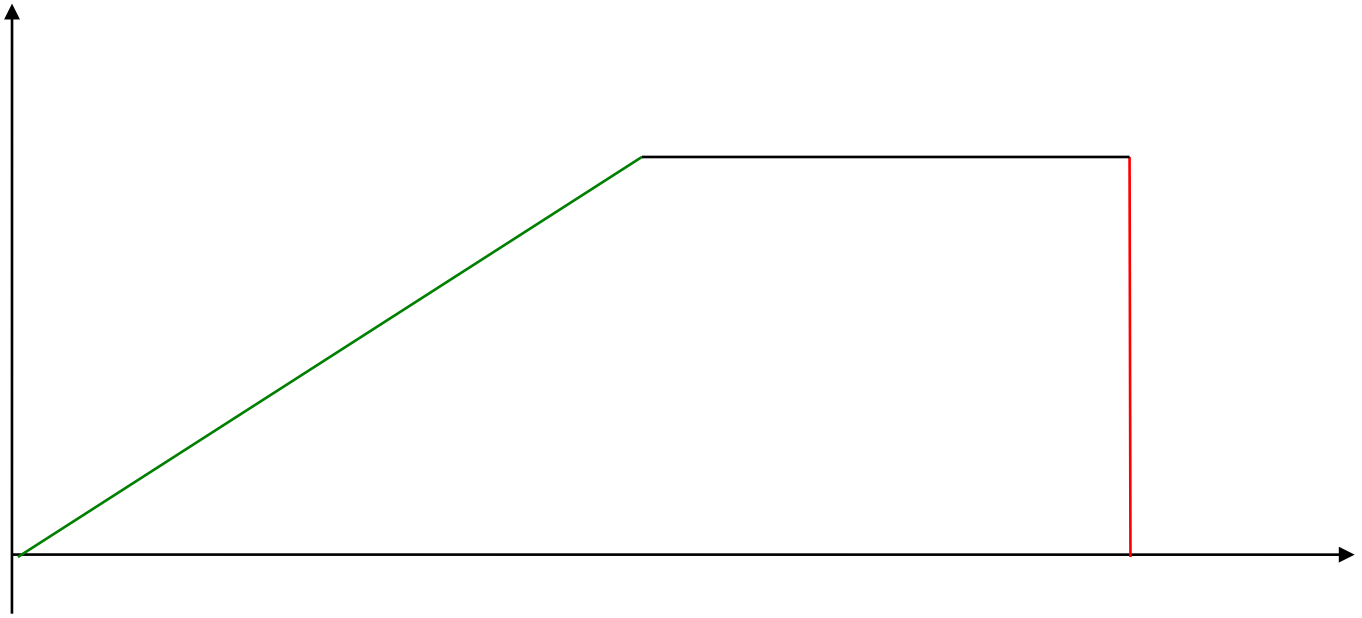
- T_1 modifiziert nacheinander die Datenobjekte *A* und *B* (z.B. eine Überweisung)
- T_2 liest nacheinander dieselben Datenobjekte *A* und *B* (z.B. zur Aufsummierung der beiden Kontostände).

Verzahnung zweier TAs gemäß 2PL

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

Strenges Zwei-Phasen Sperrprotokoll

- 2PL schließt kaskadierendes Rücksetzen nicht aus
- Erweiterung zum *strengen* 2PL:
  - alle Sperren werden bis EOT gehalten
  - damit ist kaskadierendes Rücksetzen ausgeschlossen



Verklemmungen (Deadlocks)

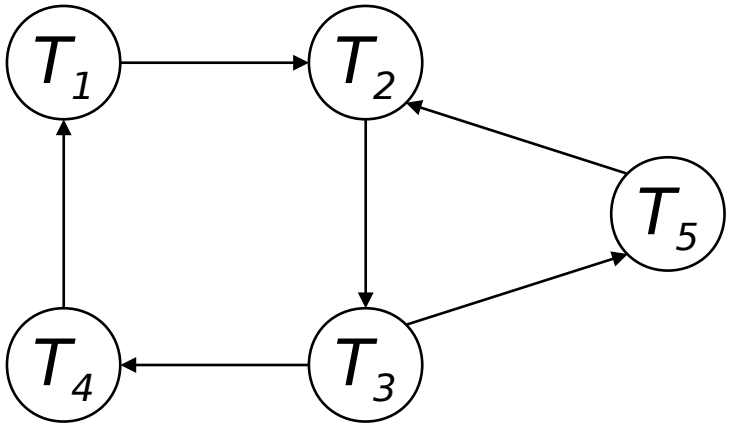
Ein verklemmter Schedule

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

Erkennungen von Verklemmungen

Wartegraph mit zwei Zyklen

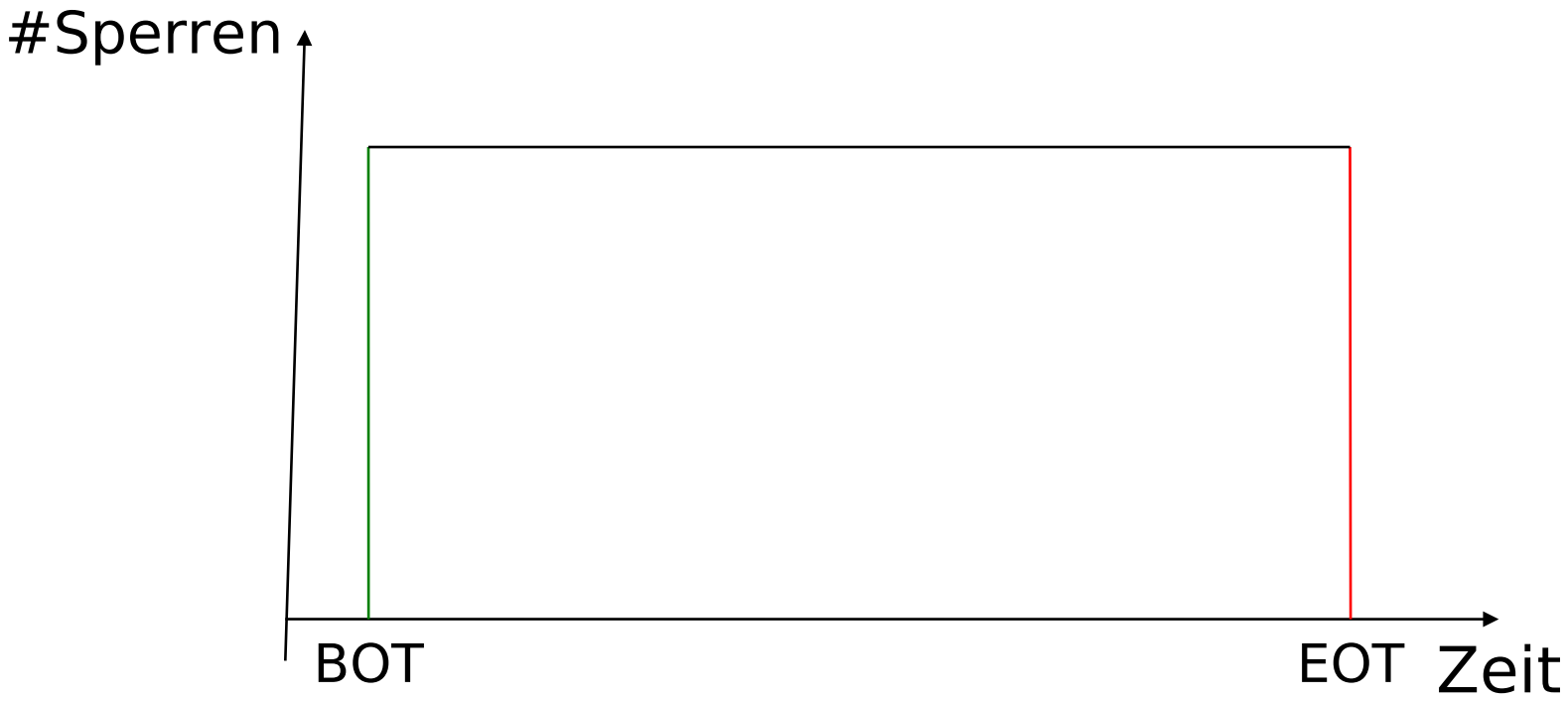
- $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$
- $T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$



- beide Zyklen können durch Rücksetzen von T_3 „gelöst“ werden
- Zyklenerkennung durch Tiefensuche im Wartegraphen

Preclaiming zur Vermeidung von Verklemmungen

Preclaiming in Verbindung mit dem strengen 2 PL-Protokoll



Praktisch nur selten einsetzbar, da i.d.R. zu Beginn einer Transaktion nicht alle zu sperrenden Tupel bekannt sind

Einfüge- und Löschoperationen, Phantome

- Vor dem Löschen eines Objekts muss die Transaktion eine *X*-Sperre für dieses Objekt erwerben. Man beachte aber, dass eine andere TA, die für dieses Objekt ebenfalls eine Sperre erwerben will, diese nicht mehr erhalten kann, falls die Löschtransaktion erfolgreich (mit **commit**) abschließt.
- Beim Einfügen eines neuen Objekts erwirbt die einfügende Transaktion eine *X*-Sperre.

Einfüge- und Löschoperationen, Phantome (2)

Löschen:

T1	T2
X(A)	
	S(A)
delete(A)	
commit	
	abort

Einfügen:

T1	T2
X(B)	
	S(B)
insert(B)	
commit	
	read(B)

Phantomprobleme

T_1	T_2
select count(*)	
from prüfen	
where Note **between** 1 **and** 2;	
	insert into prüfen
	values(19555, 5001, 2137, 1);
select count(*)	
from prüfen	
where Note **between** 1 **and** 2	

Phantomprobleme

- Das Problem lässt sich dadurch lösen, dass man zusätzlich zu den Tupeln auch den Zugriffsweg, auf dem man zu den Objekten gelangt ist, sperrt
- Wenn also ein Index für das Attribut *Note* existiert, würde der Indexbereich [1,2] für T_1 mit einer *S-Sperre* belegt
- Wenn jetzt also Transaktion T_2 versucht, das Tupel [29555, 5001, 2137, 1] in *prüfen* einzufügen, wird die TA blockiert

Zeitstempel-basierte Synchronisation

Grundidee: Prüfe, ob Historie äquivalent ist zu serieller Historie gemäß **Startzeitpunkt** der Transaktionen.

- Jeder Transaktion T_i wird ein Zeitstempel $TS(T_i)$ zugeordnet
- Jedem Datum A in der Datenbasis werden zwei Marken zugeordnet:

  1. readTS(A): Zeitstempel der *spätesten* Transaktion, die das Datum gelesen hat

  2.writeTS(A): Zeitstempel der *letzten* Transaktion, die das Datum geschrieben hat

	T1	T2	A		B		C	
Zeit	TS=1	TS=2	r	w	r	w	r	w
1	BOT		0	0	0	0	0	0
2		BOT						
3	r(A)		1					
4	w(A)			1				
5		r(A)	2					
6		w(B)				2		
7		r(C)					2	
8	r(C)							
	~~r(B)~~							
	~~w(B)~~							
	~~w(C)~~							

Äquivalenz zu serieller Historie gemäß Startzeitpunkten der TAs

Zeitstempel-basierte Synchronisation

- T_i will A lesen, also $r_i(A)$
  - Falls $TS(T_i) < writeTS(A)$ gilt, haben wir ein Problem:
    - Die Transaktion T_i ist älter als eine andere Transaktion, die A schon geschrieben hat.
    - Also muss T_i zurückgesetzt werden.
  - Anderenfalls, wenn also $TS(T_i) \geq writeTS(A)$ gilt, kann T_i ihre Leseoperation durchführen und die Marke $readTS(A)$ wird auf $max(TS(T_i), readTS(A))$ gesetzt. (späteste Transaktion, die T_i gelesen hat)

Zeitstempel-basierende Synchronisation

- T_i will A schreiben, also $w_i(A)$
  - Falls $TS(T_i) < readTS(A)$ gilt, gab es eine jüngere Lesetransaktion, die den neuen Wert von A, den T_i gerade beabsichtigt zu schreiben, hätte lesen müssen. Also muss T_i zurückgesetzt werden.
  - Falls $TS(T_i) < writeTS(A)$ gilt, gab es eine jüngere Schreibtransaktion. D.h. T_i beabsichtigt einen Wert einer jüngeren Transaktion zu überschreiben. Das muss natürlich verhindert werden, so dass T_i auch in diesem Fall zurückgesetzt werden muss.
  - Anderenfalls darf T_i das Datum A schreiben und die Marke $writeTS(A)$ wird auf $TS(T_i)$ gesetzt.

Optimistische Synchronisation

Grundidee: Überprüfe, ob Historie äquivalent ist zu serieller Historie gemäß **Validierungszeitpunkt** der Transaktionen

Optimistische Synchronisation

1. *Lesephase:*
   - In dieser Phase werden alle Operationen der Transaktion ausgeführt – also auch die Änderungsoperationen.
   - Gegenüber der Datenbasis tritt die Transaktion in dieser Phase aber nur als Leser in Erscheinung, da alle gelesenen Daten in lokalen Variablen der Transaktion gespeichert werden.
   - alle Schreiboperationen werden zunächst auf diesen lokalen Variablen aufgeführt.
2. *Validierungsphase:*
   - In dieser Phase wird entschieden, ob die Transaktion möglicherweise in Konflikt mit anderen Transaktionen geraten ist.
   - Dies wird anhand von Zeitstempeln entschieden, die den Transaktionen in der Reihenfolge zugewiesen werden, in der sie in die Validierungsphase eintreten.
3. *Schreibphase:*
   - Die Änderungen der Transaktionen, bei denen die Validierung positiv verlaufen ist, werden in dieser Phase in die Datenbank eingebracht.

Optimistische Synchronisation - Beispiele

Ta	Tj
w(a)	
	r(a)
commit	
	~~commit~~

~~Ta | Tj~~

Ta	Tj
w(a)	
	r(a)
	commit
commit	

Tj | Ta

Ta	Tj
w(a)	
commit	
	r(a)
	commit

Ta | Tj

Äquivalenz zu serieller Historie gemäß Validierungszeitpunkten der Transaktionen

Validierung bei der optimistischen Synchronisation

Vereinfachende Annahme: Es ist immer nur eine TA in der Validierungsphase!

Wir wollen eine Transaktion T_j validieren. Die Validierung ist erfolgreich falls für **alle** älteren Transaktionen T_a – also solche die früher ihre Validierung abgeschlossen haben – eine der beiden folgenden Bedingungen gelten:

1. T_a war zum Beginn der Transaktion T_j schon abgeschlossen – einschließlich der Schreibphase.
2. Die Menge der von T_a geschriebenen Datenelemente, genannt $WriteSet(T_a)$ enthält keine Elemente der Menge der gelesenen Datenelemente von T_j, genannt $ReadSet(T_j)$. *Es muss also gelten:*

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

Transaktionsverwaltung in SQL92

set transaction

[read only, |read write,]

[**isolation level**

read uncommitted, |

read committed, |

repeatable read, |

serializable,]

[**diagnostic size** ...,]

Transaktionsverwaltung in SQL92

- **read uncommitted:** Dies ist die schwächste Konsistenzstufe. Sie darf auch nur für **read only-**Transaktionen spezifiziert werden. Eine derartige Transaktion hat Zugriff auf noch nicht festgeschriebene Daten. Zum Beispiel ist folgender Schedule möglich:

T_1	T_2
	read(A)
	. . .
	write(A)
read(A)	
. . .	
	rollback

Transaktionsverwaltung in SQL92

- **read committed:** Diese Transaktionen lesen nur festgeschriebene Werte. Allerdings können sie unterschiedliche Zustände der Datenbasis-Objekte zu sehen bekommen:

T_1	T_2
read(A)	
	write(A)
	write(B)
	commit
read(B)	
read(A)	
. . .	

Transaktionsverwaltung in SQL92

- **repeatable read:** Das oben aufgeführte Problem des *non repeatable read* wird durch diese Konsistenzstufe ausgeschlossen. Allerdings kann es hierbei noch zum Phantomproblem kommen. Dies kann z.B. dann passieren, wenn eine parallele Änderungstransaktion dazu führt, dass Tupel ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.
- **serializable:** Diese Konsistenzstufe fordert die Serialisierbarkeit. Dies ist der Default.