Kapitel 12

Dr. Jérôme Kunegis

Transaktionsverwaltung

WeST –
Institut für Web Science & Technologien

Lernziele

- Begriff und Eigenschaften von Transaktionen
- Mehrbenutzer-Synchronisation
  - Theorie der Serialisierbarkeit
  - Sperrprotokolle
  - Deadlocks und deren Vermeidung

- Zusammenfassung & Ausblick

Transaktionsverwaltung

Beispiel einer typischen Transaktion in einer Bankanwendung:

1. Lese den Kontostand von *A* in die Variable *a* : **read**(*A*,*a*);
2. Reduziere den Kontostand um 50 EUR: *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 EUR: *b* := *b* + 50;
6. Schreibe den neuen Kontostand in die Datenbasis: **write**(*B*, *b*);

Operationen auf Transaktions-Ebene

In den klassischen Transaktionssystemen:

- **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**.

1. Den erfolglosen Abschluss durch ein **abort**.

Eigenschaften von Transaktionen: ACID

Atomicity (Atomarität)

Alles oder nichts !

Consistency

Konsistenter DB-Zustand muss in konsistenten Zustand übergehen !

Isolation

Jede Transaktion hat die DB „für sich allein“

Durability (Dauerhaftigkeit)

Änderungen erfolgreicher Transaktionen dürfen nie verloren gehen

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.

ROLLBACK WORK

Alle Änderungen sollen zurückgesetzt werden. Anders als der **COMMIT**-Befehl muss das DBMS die „*erfolgreiche*“ Ausführung eines rollback-Befehls immer garantieren können.

Transaktionsverwaltung in SQL

Beispielsequenz auf Basis des Universitätsschemas:

```
INSERT INTO Professoren
        VALUES (2141, 'Meitner', 'C4', 205);
INSERT INTO Vorlesungen
        VALUES (5275, 'Kernphysik', 3, 2141);
COMMIT;
```

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.		$a_2 := a_2 - 200$
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

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.		$a_2 := a_2 - 200$
6.		write(A,a_2)
7.	read(B,b_1)	
8.	. . .	
9.	abort	

Fehler bei unkontrolliertem Mehrbenutzerbetrieb III

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**	

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**	

Ist nicht serialisierbar, obwohl dies im konkreten Fall nicht zum Konflikt führt. Letzteres kann die DB aber nicht beurteilen.

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**	

Dieselbe r/w-Konstellation, diesmal zum Konflikt führend. Die DB greift zur Kontrolle nur auf die r/w-Struktur zu, nicht auf die Semantik der Anwendung!

Theorie der Serialisierbarkeit

Ziel: „Formale“ Beschreibung 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 **ABORT**s,
- 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

Historie

$r_i(A)$ 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 ist eine partiell geordnete Menge $(H, <_H)$ mit

$$H = \bigcup_i T_i$$

$<_H$ ist verträglich mit allen $<_i$ - *Ordnungen, d.h.:*

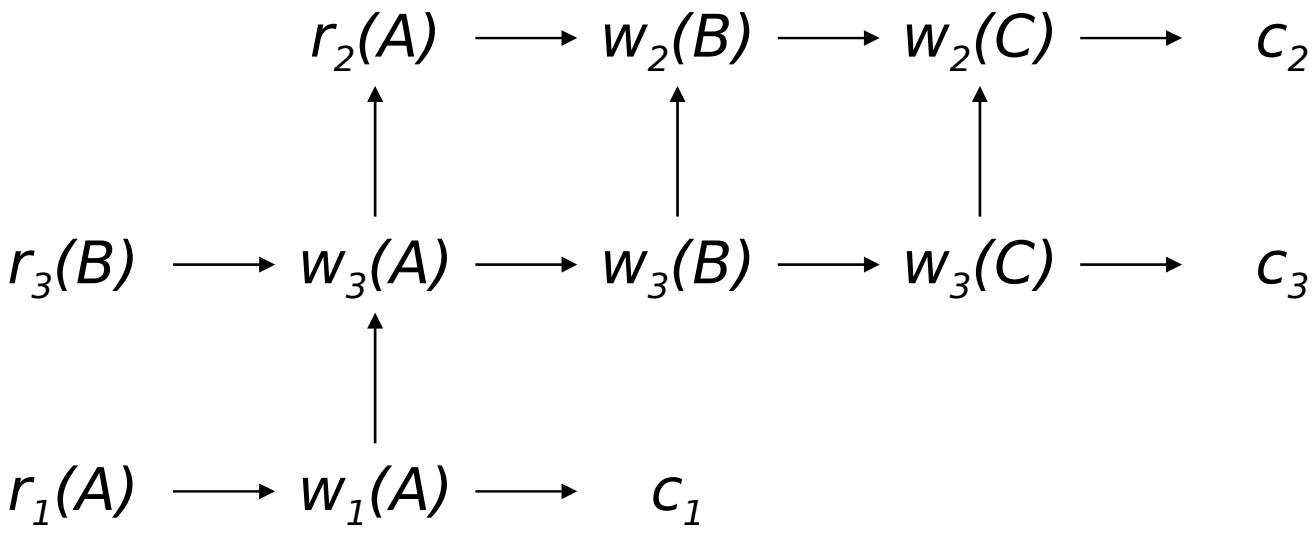
$$p <_i q \Rightarrow p <_H q$$

Die Ordnung muss nicht total sein, z.B. bei Mehrprozessorbetrieb.

Für zwei Konfliktoperationen $p,q \in H$ gilt entweder

- $p <_H q$ oder
- $q <_H p$.

Beispiel-Historie für drei Transaktionen



Äquivalenz zweier Historien

H ≡ H' wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen.

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**

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**

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**

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**

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

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

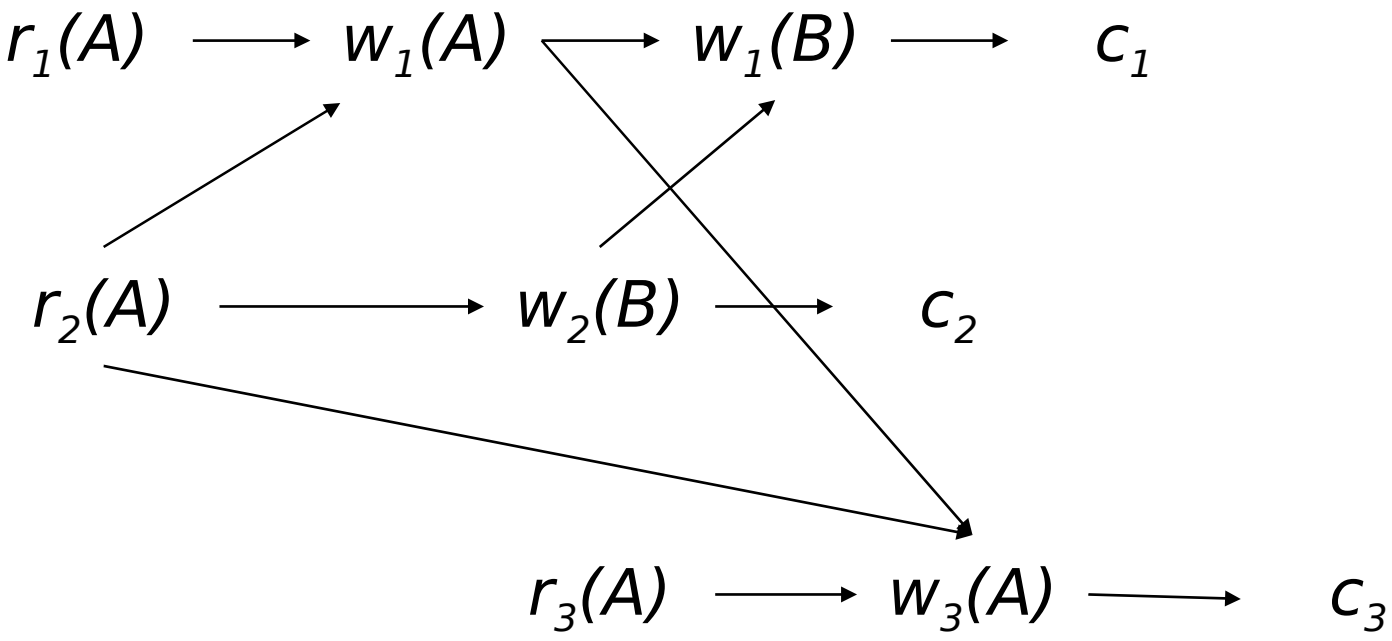
$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$

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

Serialisierbare Historie

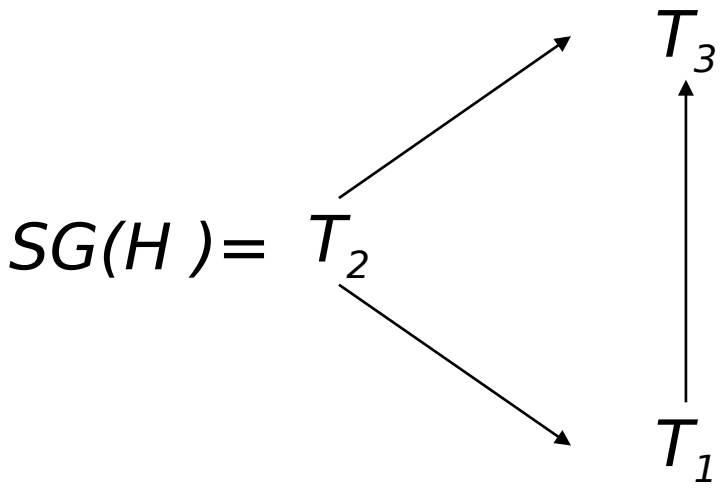
Eine Historie ist *serialisierbar,* wenn sie äquivalent zu einer seriellen Historie Hs ist.

Historie ...



Serialisierbarkeitsgraph

… und zugehöriger 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

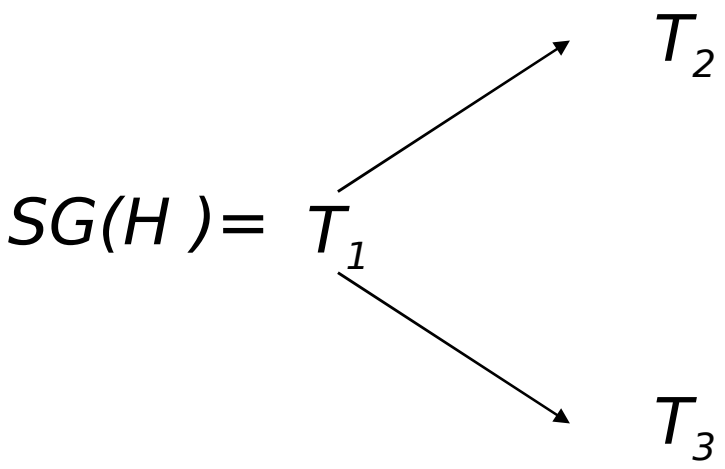
Serialisierbarkeitstheorem

Satz: 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)

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

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

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

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 Recovery

Beispiel-Historie mit kaskadierendem Rücksetzen:

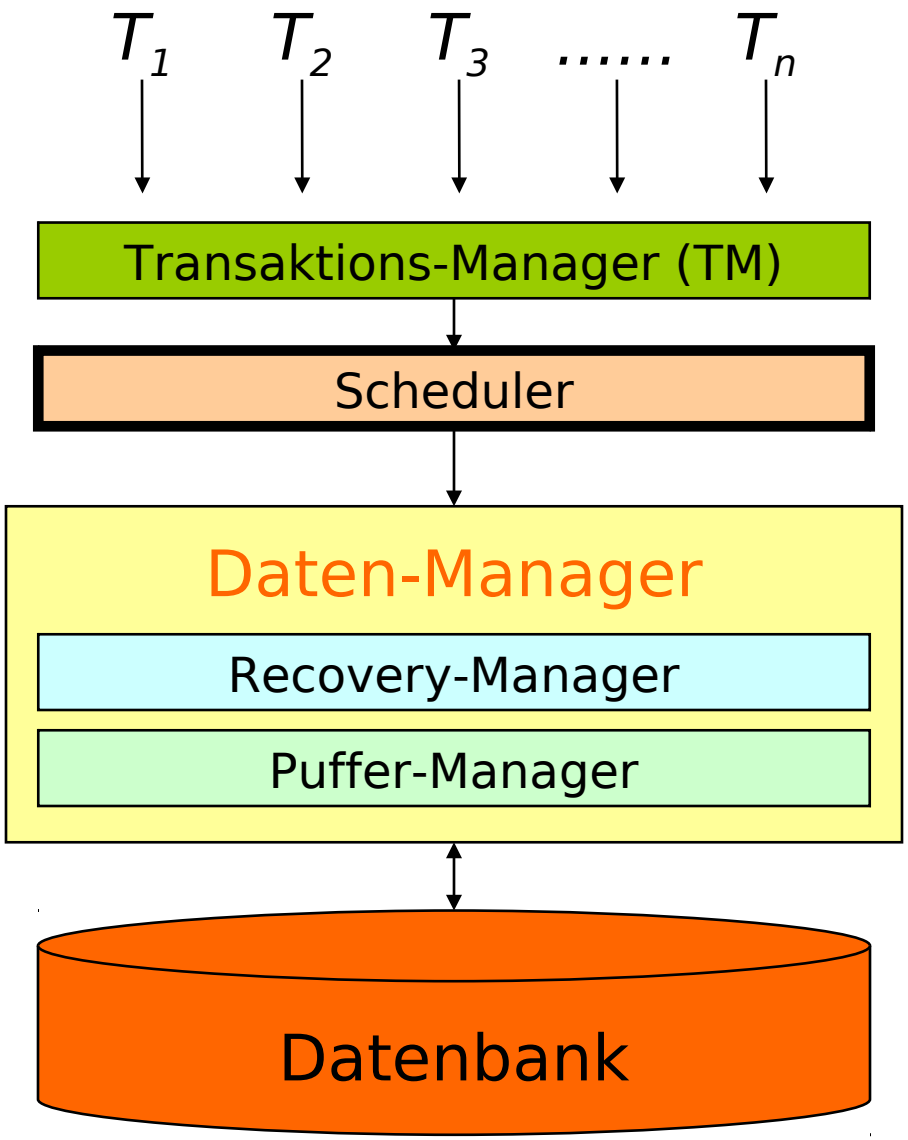
Schritt	T_1	T_2	T_3	T_4	T_5
0.	. . .				
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_4(D)$	
8.					$r_5(D)$
9.	a_1(**abort**)				

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 liest, das zuvor von T_j geschrieben wurde

Der Datenbank-Scheduler



Aufgabe des Schedulers

- Reihung der Operationen verschiedener Transaktionen, so dass die Historie
  - mindestens serialisierbar
  - und meistens auch ohne kaskadierendes Rollback rücksetzbar ist.

- Verschiedene Ansätze möglich:
  - sperrbasierte Synchronisation (am häufigsten)
  - Zeitstempel-basierte Synchronisation

  } Pessimistische Synchronisation

  - optimistische Synchronisation (bei vorwiegend lesenden Zugriffen)

Sperrbasierte Synchronisation

Sichert Serialisierbarkeit zu

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

Jedes Objekt, das von einer Transaktion benutzt werden soll, muss vorher entsprechend gesperrt werden.

Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.

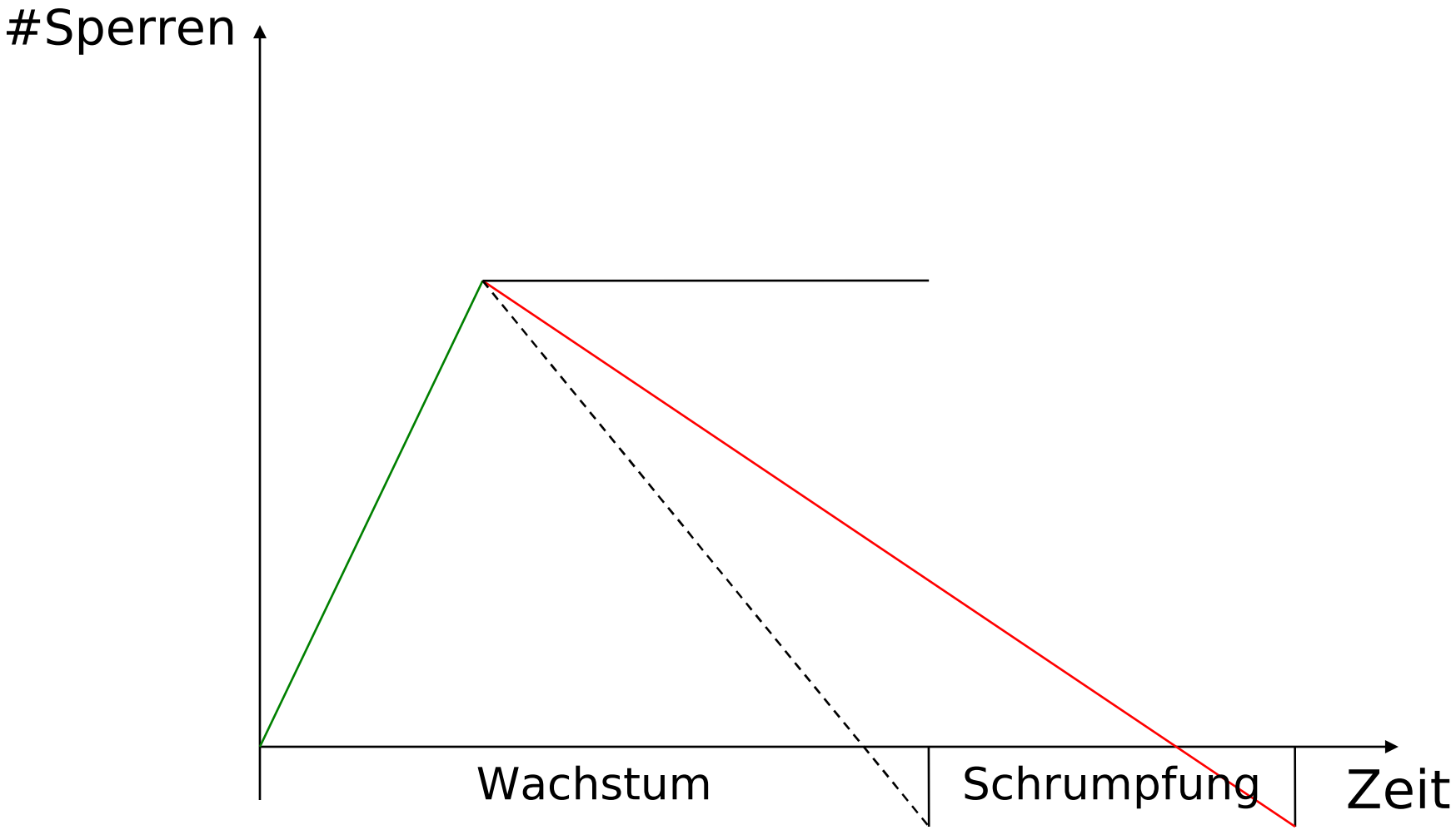
Eine Transaktion muss die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht – bis die Sperre gewährt werden kann.

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.

Bei 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

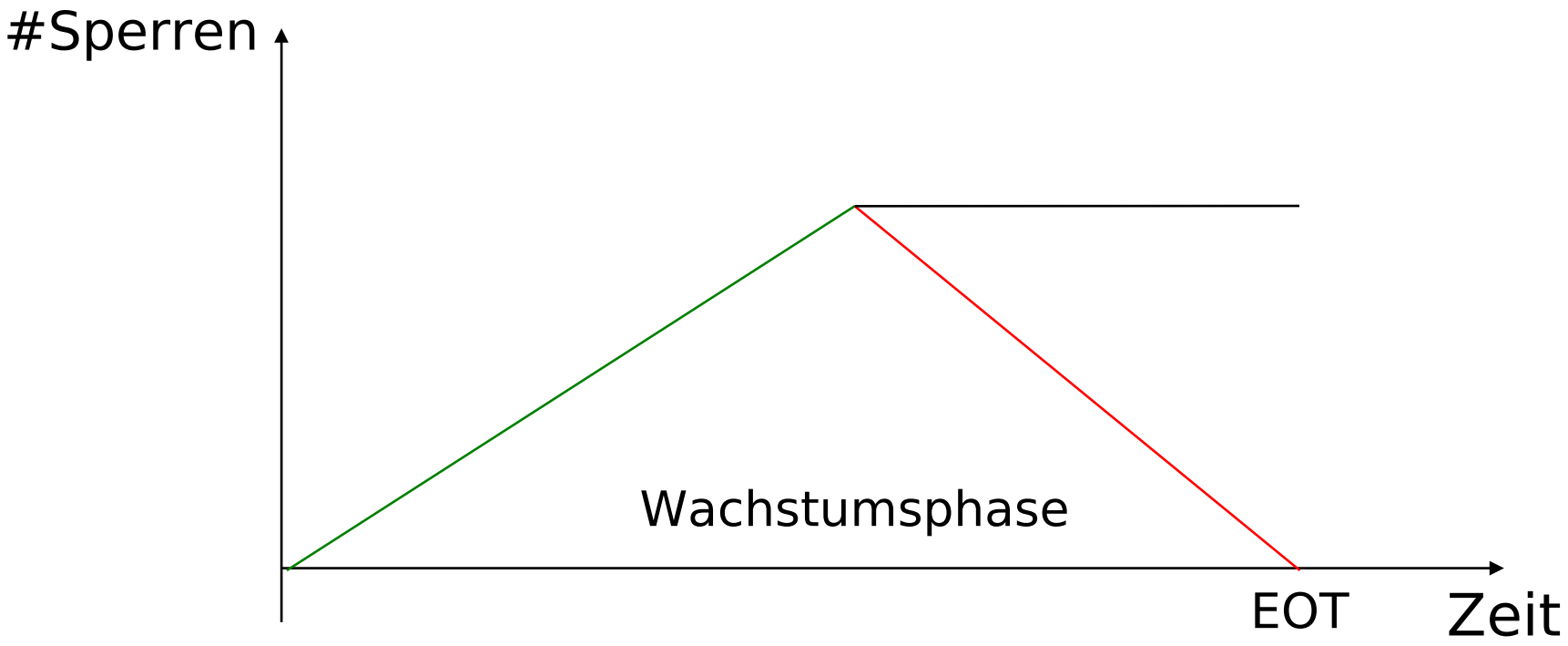
Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(*A*)		
3.	read(*A*)		
4.	write(*A*)		
5.		**BOT**	
6.		**lockS**(*A*)	T_2 muss warten
7.	**lockX**(*B*)		
8.	read(*B*)		
9.	**unlockX**(*A*)		T_2 wecken
10.		read(*A*)	
11.		**lockS**(*B*)	T_2 muss warten
12.	write(*B*)		
13.	**unlockX**(*B*)		T_2 wecken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

Verklemmungen

- a: können entdeckt und dann aufgelöst
- b: oder gleich vermieden werden

Beides ist u.U. teuer. Mögliche Techniken:

Zu a:

1. Time-Out
2. Zyklenerkennung

Zu b:

3. Preclaiming
4. Zeitstempel

Erkennen von Verklemmungen

1. Brute-Force-Methode: Time-Out

- **Nach gewisser Wartezeit (z.B. 1 sec) wird Transaktion zurückgesetzt**
- **Nachteil: Falls Zeit zu kurz, werden zu viele Transaktionen zurückgesetzt, die nur auf Resourcen (CPU etc.) warten. Falls Zeit zu lang, werden zu viele Verklemmungen geduldet.**

Erkennen von Verklemmungen

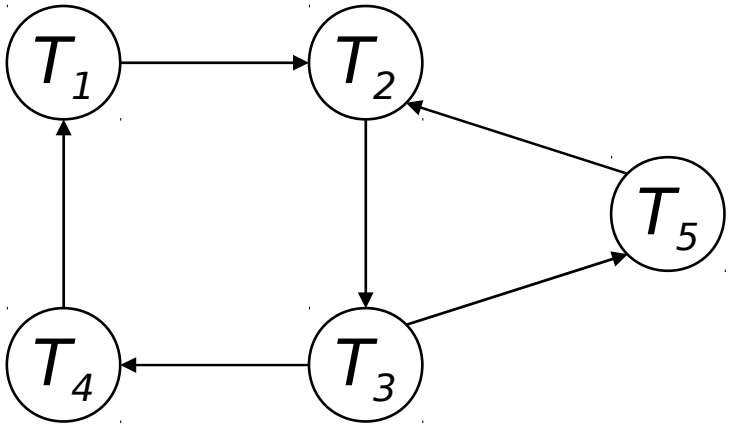
2. Zyklenerkennung durch Tiefensuche im Wartegraph

ist aufwendiger, aber präziser

Bsp.: Wartegraph mit zwei Zyklen:

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$

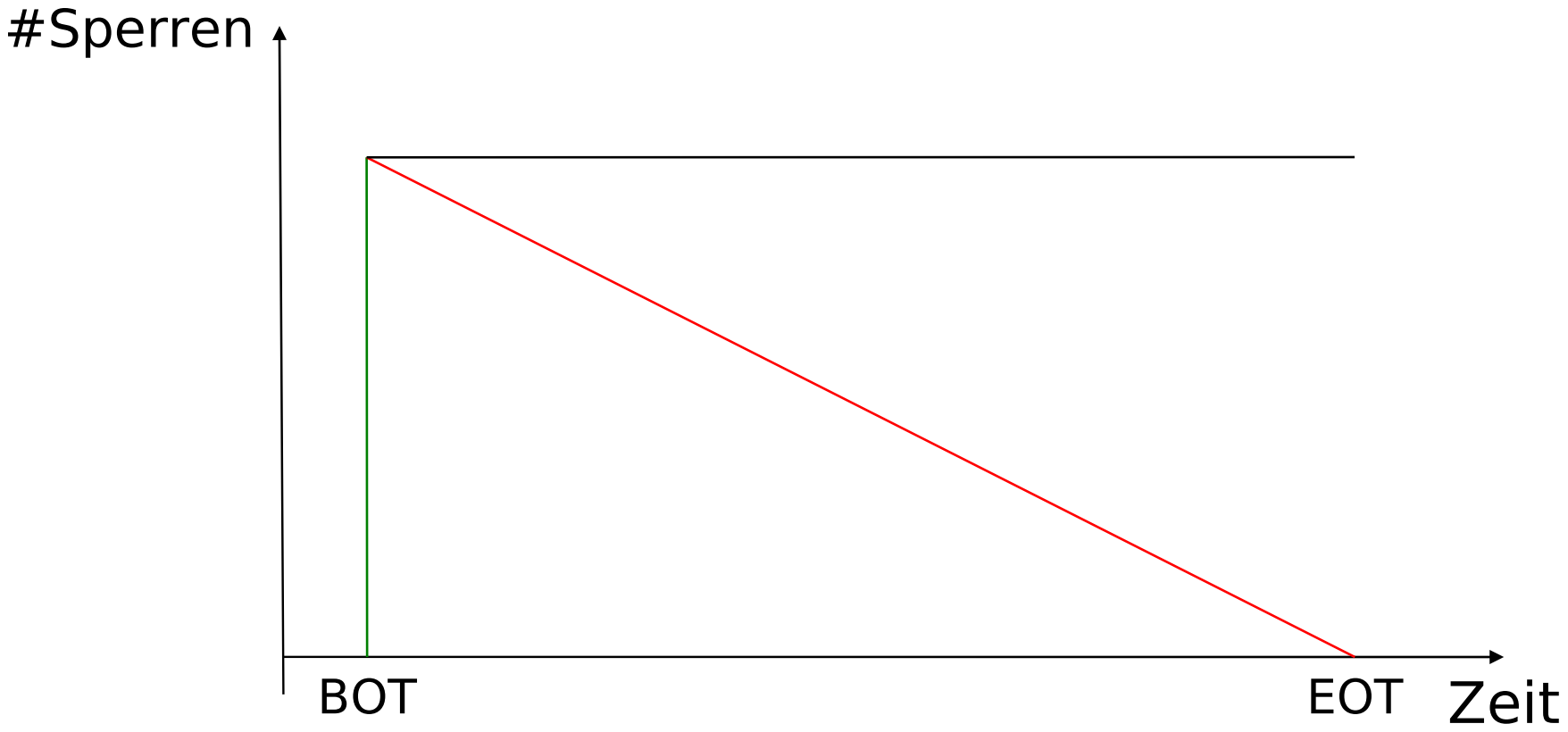
$T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$



Beide Zyklen können durch Rücksetzen von T_3 „gelöst“ werden.

Vermeiden von Verklemmungen

3. Preclaiming in Verbindung mit dem strengen 2 PL-Protokoll



Preclaiming

ist in der Praxis häufig nicht einzusetzen:

Man weiss i.A. apriori nicht genau, welche Sperren benötigt werden, z.B. bei if...then...else im Anwendungsprogramm.

Daher müssen normalerweise mehr Sperren als nötig „auf Verdacht“ reserviert werden, was zu übermäßiger Resourcenbelegung und Einschränkung der Parallelität führt.

Vermeiden von Verklemmungen

4. Verklemmungsvermeidung durch Zeitstempel

Jeder Transaktion wird ein eindeutiger Zeitstempel (TS) zugeordnet

ältere TAs haben einen kleineren Zeitstempel als jüngere TAs

TAs dürfen nicht mehr „bedingungslos“ auf eine Sperre warten.

Zwei Varianten:

wound-wait Strategie

T_1 will Sperre erwerben, die von T_2 gehalten wird.

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.

wait-die Strategie

T_1 *will Sperre erwerben, die von* T_2 gehalten wird.

Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre.

Sonst wird T_1 abgebrochen und zurückgesetzt.

Geschafft !