Karlsruher Institut für Technologie

Datenbanksysteme

Kapitel 9: Nebenläufigkeit und Transaktionen

Lehrstuhl für Systeme der Informationsverwaltung, Fakultät für Informatik

Photo by Mikko Karvonen

Eigenschaften von Transaktionen

Karlsruher Institut für Technologie

- Kernkonzept zur Sicherstellung der korrekten Funktionsweise im **Mehrbenutzerbetrieb**
- Programm als Folge von Lese- und Schreiboperationen, die eine Datenbank von einem konsistentem Zustand in einen (möglicherweise) geänderten Zustand überführen, wobei das **ACID** Prinzip gilt.
  - Atomicity: Transkation wird vollständig oder gar nicht ausgeführt
  - Consistency (Integritätserhaltung) – Vor und nach der Transaktion gelten sämtlich Integritätsbedingungen
  - Isolation – Nutzer hat Eindruck er arbeite allein auf der Datenbank
  - Durability – Ergebnis wird dauerhaft gespeichert

Beispiel Probleme ohne Transaktionen

Karlsruher Institut für Technologie

- Isolation: Zwischenzustände sind für andere Nutzer nicht sichtbar
- Atomariät: Ganze Überweisung ausführen, oder gar nichts
  - Beispiel, „Bank-Szenario“:

Nummer	Inhaber	Stand
	Klemens	5000
	Gunter	200

  - Überweisung – zwei Elementaroperationen + Trankaktionskommandos (BEGIN, COMMIT, ABORT)
    1. BEGIN
    2. **Abbuchung(Klemens, 500),**
    3. **Einzahlung(Gunter, 500),**
    4. COMMIT

Autocommit Ausstellen!

Einleitung/ Probleme
Definitionen
Locking
Dienste

Synchronisation in Datenbanken (1)

Karlsruher Institut für Technologie

- Zentrales Leistungsmerkmal von Datenbanken:
  Viele Benutzer können die gleichen Daten gleichzeitig sowohl lesend als auch schreibend zugreifen.
- Konsistenz muss sichergestellt sein –
  Aufgabe der **Synchronisationskomponente**.
- Benutzer sollen vom Multi-User Betrieb so wenig wie möglich merken.
  Nebenläufigkeit soll transparent sein.
  ‚Illusion', dass man der einzige Nutzer ist.
- Beispiel hat dies illustriert (bzw. das Gegenteil davon).
- Engl. *concurrency*, *concurrency control*.

Synchronisation in Datenbanken (2)

Karlsruher Institut für Technologie

- Einfache korrekte Lösung: Alles seriell ausführen
- **Serielle Ausführung** von Anwendungsprogrammen.
  - Jene Illusion lässt sich
    ohne jeglichen Synchronisationsaufwand erreichen.
  - Datenbank ist nach Ende jeder Transaktion konsistent.
- **Aber**:
  - Extreme Wartezeiten
  - unbefriedigende Ausnutzung der Ressourcen.
    (CPU ist während Kommunikation und I/O nicht aktiv.)

Synchronisation im Allgemeinen

- Das andere Extrem: Unkontrollierte **nicht-serielle Ausführung**

- Führt zu mehreren Problemen:
  - Lost Updates,
  - inkonsistente Lesezugriffe („non-repeatable read“),
  - Dirty Reads, d. h. Reads von Updates, die noch nicht committet sind.
  - Phantome.

Lost Update

- Programm T_1 transferiert EUR 300,- von Konto A auf Konto B, Programm T_2 schreibt Konto A 3 % Zinsen gut.
- Zinsen aus Schritt 5 von Programm T_2 gehen verloren, weil T_1 den Wert in Schritt 6 überschreibt.

Schritt	T1	T2
1	Read(A, a1)	
2	a1 := a1-300	
3		Read(A, a2)
4		a2 := a2 *1.03
5		Write(A, a2)
6	Write(A, a1)	
7	Read(B, b1)	
8	b1 := b1 + 300	
9	Write(B, b1)	

Dirty Read

- *Commit*, *Abort*.
- Programm T_2 schreibt Zinsen gut, basierend auf einem Wert, der nicht zu einem konsistenten Zustand gehört.
- Denn später erfolgt **abort** von T_1.

Schritt	*T1*	*T2*
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 *1.03
6		Write(A, a2)
7		**commit**
8	Read(B, b1)	
9	…	
10	**abort**	

Dirty Read – Anmerkungen

- Abort nicht nur, weil User/Anwendung es so will.
  DBMS kann Transaktion aborten, um Isolation sicherzustellen.
- Dieses Kapitel geht nur kurz am Ende
  auf Maßnahmen gegen Dirty Reads ein.
  Recoverability schließt Dirty Reads aus.

Non-Repeatable Reads (Mehrbenutzeranomalie)

Karlsruher Institut für Technologie

Programm liest Datenobjekt mehr als einmal
und sieht Änderung, die anderes Programm durchgeführt hat.

Schritt	*T1*	*T2*
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 *1.03
6		Write(A, a2)
7	Read(A, a3)	
8	…	

Einleitung/
Probleme
Definitionen
Locking
Dienste

Z

Phantom Problem

Karlsruher Institut für Technologie

Berechnung von Änderungen auf veralteten Werten, hier Mitarbeiteranzahl.

T_1	T_2
select count (*) **into** X **from** employee **where** dept='CS';	
	insert into employee **values** (6789, 'Lilo', 'Pause', 'CS'); **commit**;
update employee **set** Bonus = Bonus+10000/X **where** dept='CS'; **commit**;	

Z

Transaktionen (1)

- *Transaktion* := Folge von reads und writes auf Datenobjekten (z.B. Tupel).
- Programmausführung ≠ Programm-Code
- Genauer: Repräsentation der Ausführung, die folgende Charakteristika umfaßt:
  - Gelesene und geschriebene Datenobjekte,
  - Reihenfolge ihrer Ausführung,
  - kommt am Ende ein Commit (bzw. Abort) oder nicht?

Transaktionen (2)

Karlsruher Institut für Technologie

- Beispiel:

```
Procedure P begin
        Start Transaction;
        temp := Read(x);
        temp := temp + 1;
        Write(x, temp);
        Commit;
end
```

- Repräsentation: $r_1[x] \rightarrow w_1[x] \rightarrow c_1$
- Transaktion ist **partielle Ordnung**
  - Vollständige Reihenfolge der Operationen muss nicht bekannt sein
  - Erzeugt Freiheitsgrade beim Anordnen
  - Besonders wichtig wenn mehre Transaktionen betrachtet werden

Konflikt – Wann treten Probleme auf?

Karlsruher Institut für Technologie

- *Zwei Operationen p, q konfligieren* :=
  p, q greifen auf das **gleiche Datenobjekt** zu,
  und p oder q ist eine **Schreiboperation**.
- Weitere Operationen –
  Definition von 'Konflikt' muss erweitert werden.
- Beispiel. Kompatibilitätsmatrix: (Konfliktmatrix ist invers dazu)

	Read	Write
Read	**y**	**n**
Write	**n**	**n**

Einleitung/ Probleme

Definitionen

Locking

Dienste

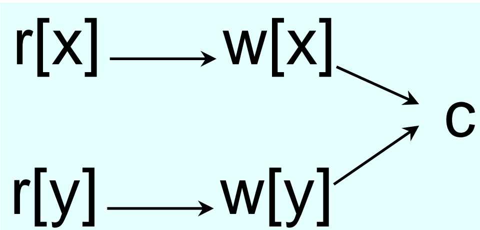
Transaktion – formale Definition (1)

Transaktion ist partielle Ordnung mit Ordnungsrelation $<$, so dass gilt

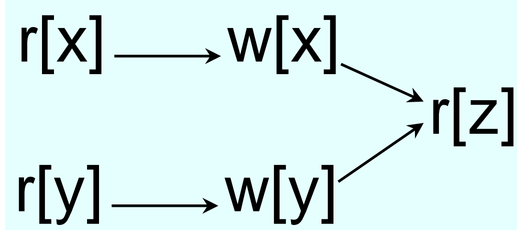
1. $T \subseteq \{r[x], w[x] | x \text{ ist ein Datenobjekt}\} \cup \{a, c\}$,
2. $a \in T \Leftrightarrow c \notin T$;
3. wenn es sich bei t um c oder a handelt, dann gilt für jede andere Operation $p \in T$: $p < t$; und
4. wenn $r[x], w[x] \in T$, dann $r[x] < w[x]$ oder $w[x] < r[x]$.

Transaktion – formale Definition (2)

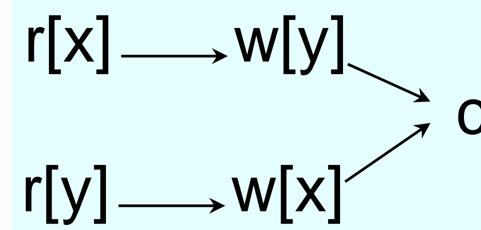
Beispiele:



ist Transaktion.



ist keine TA.



ist keine TA.

Einleitung/ Probleme
Definitionen
Locking
Dienste

Histories – Verzahnung von Transaktionen

Karlsruher Institut für Technologie

- Ausführung der Operationen mehrerer Transaktionen, die miteinander 'verzahnt' sind, d. h. nebenläufig ablaufen.
- Formal –
  $T = \{T_1, T_2, \ldots, T_n\}$ ist Menge von Transaktionen.
- ***Vollständige Historie*** *H über T* :=
  partielle Ordnung mit Ordnungsbeziehung $<_H$,
  so dass

1. $H = \bigcup_{i=1}^{n} T_i$ Keine Operationen „vergessen"
2. $<_H \supseteq \bigcup_{i=1}^{n} <_i$ Partielle Ordnung aller beteiligten Transkationen sind enthalten

$p, q \in H$ konfligieren $\Rightarrow p <_H q$ oder $q <_H p$

➢ *Historie* := Präfix einer vollständigen Historie

Einleitung/ Probleme

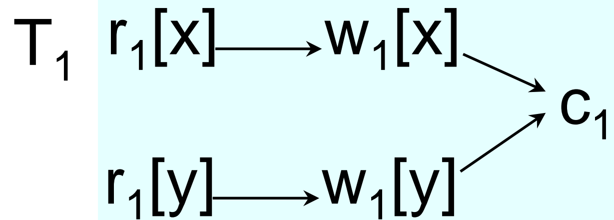
Definitionen

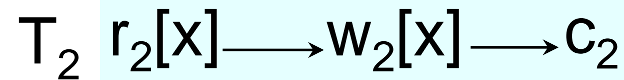
Locking

Dienste

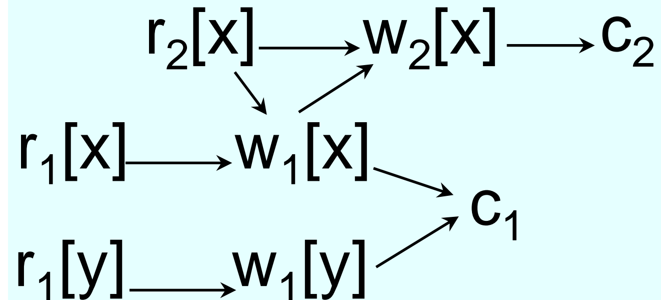
Histories – Beispiele (1)

- Gegeben zwei Transaktionen:

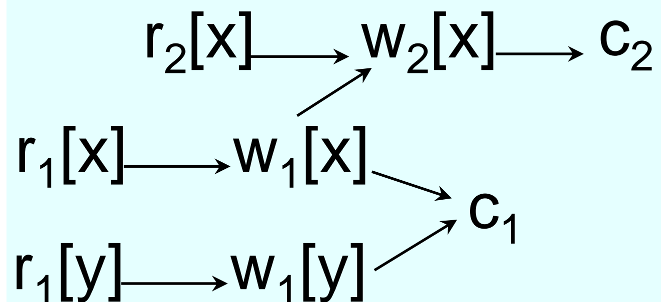


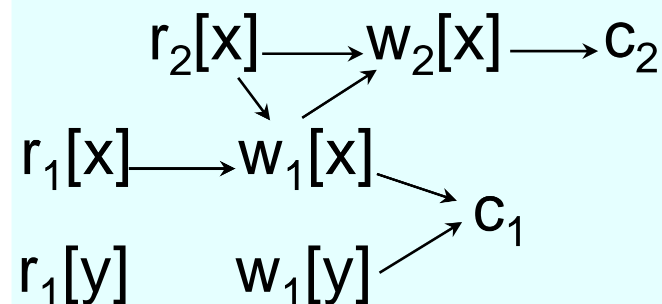


- Eine vollständige History, OK:



- Keine Histories:





Einleitung/ Probleme

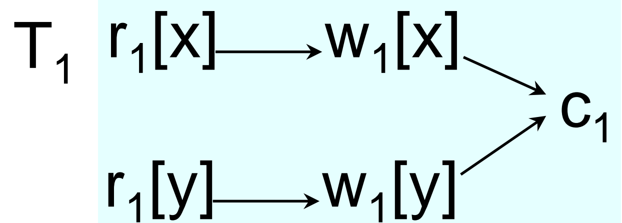
Definitionen

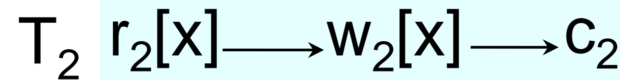
Locking

Dienste

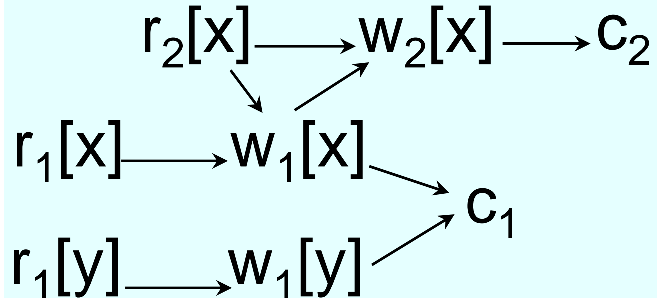
Histories – Beispiele (1)

- Gegeben zwei Transaktionen:

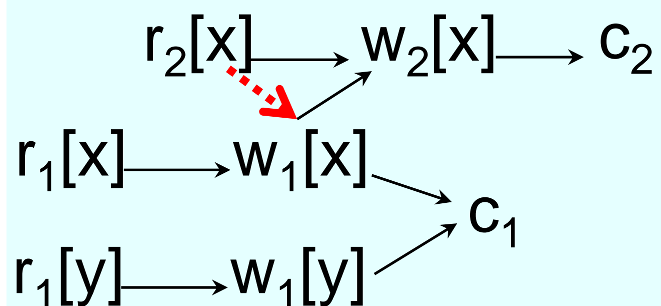


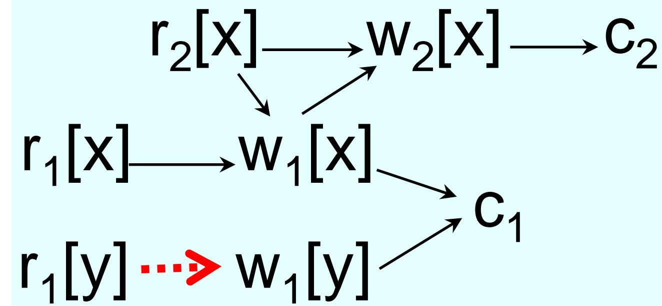


- Eine vollständige History, OK:



- Keine Histories:





Einleitung/ Probleme

Definitionen

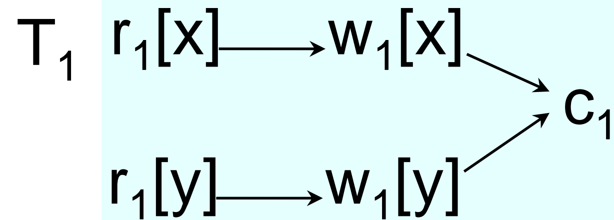
Locking

Dienste

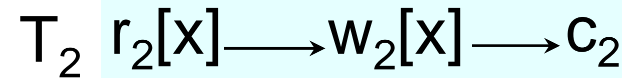
Histories – Beispiele (2)

- Gegeben zwei Transaktionen:

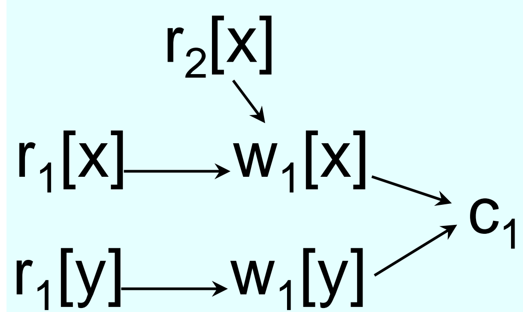
T_1



T_2



- History, aber keine vollständige History:



Einleitung/ Probleme

Definitionen

Locking

Dienste

Histories (3)

- History muss nicht korrekt sein,
  kann z. B. Lost Updates etc. enthalten.

Schritt	*T1*	*T2*
1	Read(A, a1)	
2	a1 := a1-300	
3		Read(A, a2)
4		a2 := a2 *1.03
5		Write(A, a2)
6	Write(A, a1)	
7	Read(B, b1)	
8	b1 := b1 + 300	
9	Write(B, b1)	

Ist History.

- Ziel im folgenden:
  Definition 'Korrektheit' für Histories.

Karlsruher Institut für Technologie

Serialiserbarkeit

Wann sind Histories korrekt? Und was heißt eigentlich korrekt?

Begriffsbildung: Korrektheit

T_1 : $\mathbf{read}(A)$; $A := A - 10$; $\mathbf{write}(A)$; $\mathbf{read}(B)$;
$B := B + 10$; $\mathbf{write}(B)$;

T_2 : $\mathbf{read}(B)$; $B := B - 20$; $\mathbf{write}(B)$; $\mathbf{read}(C)$;
$C := C + 20$; $\mathbf{write}(C)$;

Execution 1		Execution 2		Execution 3	
T_1	T_2	T_1	T_2	T_1	T_2
read(A)		read(A)		read(A)	
$A - 10$			read(B)	$A - 10$	
write(A)		$A - 10$			read(B)
read(B)			$B - 20$	write(A)	
$B + 10$		write(A)			$B - 20$
write(B)			write(B)	read(B)	
	read(B)	read(B)			write(B)
	$B - 20$		read(C)	$B + 10$	
	write(B)	$B + 10$			read(C)
	read(C)		$C + 20$	write(B)	
	$C + 20$	write(B)			$C + 20$
	write(C)		write(C)		write(C)

Einleitung/ Probleme

Definitionen

Locking

Dienste

Alle Transaktionen werden committen.

Serialisierbarkeit als Teil der Korrektheit (1)

Karlsruher Institut für Technologie

- Idee: Effekt einer History soll dem **einer** seriellen Ausführung entsprechen
  - Eigentlich nie vollständige serielle History vorhanden
  - **Partielle Histories -** Mischung aus committeten und nicht-comitteten Transaktionen
  - Serialisierbarkeit := Äquivalenz der committed projection

- **Committed projection** einer History H –
  Abkürzung: C(H) := resultiert aus H,
  indem alle Operationen gelöscht werden, die nicht committed sind.
- H ist **serialisierbar**,
  wenn C(H) zu serieller History H_S **äquivalent** ist.
- Abgeschlossenheitseigenschaft: Effekt gilt auch für jeden Präfix der C(H): **prefix commit-closed**

Serialisierbarkeit - Äquivalenz

Karlsruher Institut für Technologie

- Effekt-Äquivalenz?

Execution 1		Execution 2		Execution 3	
T_1	T_2	T_1	T_2	T_1	T_2
read(A)	seriell	read(A)		read(A)	
$A-10$			read(B)	$A-10$	
write(A)		$A-10$			read(B)
read(B)			$B-20$	write(A)	
$B+10$		write(A)			$B-20$
write(B)			write(B)	read(B)	
	read(B)	read(B)			write(B)
	$B-20$		read(C)	$B+10$	
	write(B)	$B+10$			read(C)
	read(C)		$C+20$	write(B)	
	$C+20$	write(B)			$C+20$
	write(C)		write(C)		write(C)

Alle Transaktionen werden committen.

	A	B	C	$A+B+C$
Initial Value	10	10	10	30
After execution 1	0	0	30	30
After execution 2	0	0	30	30
After execution 3	0	20	30	50

Integritätsbedingung Summe = 30

Einleitung/ Probleme

Definitionen

Locking

Dienste

Äquivalenz von Histories

Karlsruher Institut für Technologie

- Effekt-Äquivalenz am allgemeingültigsten, aber nicht sinnvoll
  - Ergebnis (aller) seriellen Ausführung muss bekannt sein
- Weitere Definitionen, die auf partiellen Histories berechnet werden kann:
  - **(Konflikt-)Äquivalenz (CSR)**,
  - (Sicht-)Äquivalenz (VSR).
- Im Folgenden nur Konflikt-Äquivalenz.
- Definition Konflikt-Äquivalenz':
  Histories H, H' sind (Konflikt-)äquivalent, wenn

1. gleiche Transaktionen, gleiche Operationen;
2. gleiche Ordnung konfligierender Operationen: D.h. Gleiche Konflikrelationen `conf()`

Konflikt-Äquivalenz: Konfliktrelation

Karlsruher Institut für Technologie

- Für alle Paare von Transaktionen in der History bestimme konfligierende Operationen
  - Zugriff auf gleiches Datenobjekt durch unterschiedliche Transaktion
  - Mindestens eine der Operationen ist ein Schreibzugriff
- Beispiel
  - $H_1 = r_1[x] \rightarrow r_1[y] \rightarrow w_2[x] \rightarrow w_1[y] \rightarrow r_2[z] \rightarrow w_1[x] \rightarrow w_2[y] \rightarrow c_1 \rightarrow c_2$
  - $H_2 = r_1[y] \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow w_2[x] \rightarrow w_1[x] \rightarrow r_2[z] \rightarrow w_2[y] \rightarrow c_1 \rightarrow c_2$

➢ $conf(H_1) = \{(r_1[x],w_2[x]), (w_2[x],w_1[x]), (r_1[y],w_2[y]), (w_1[y],w_2[y])\}$

➢ $conf(H_2) = \{(r_1[x],w_2[x]), (w_2[x],w_1[x]), (r_1[y],w_2[y]), (w_1[y],w_2[y])\}$

Konflikt-Äquivalenz: Konfliktrelation (2)

Karlsruher Institut für Technologie

- History 1:

Schritt	T1	T2	T3
1	Read(A)		
2		Write(A)	
3	Write(A)		
4			Write(A)

Alle Transaktionen werden committen. (Unten ebenso.)

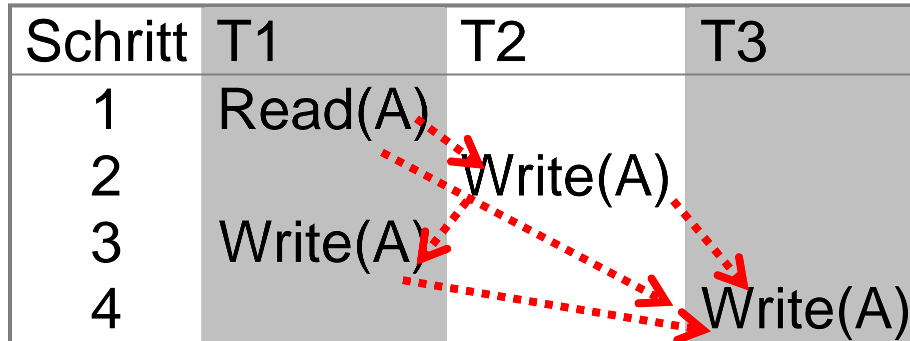
- History 2:

Schritt	T1	T2	T3
1	Read(A)		
2	Write(A)		
3		Write(A)	
4			Write(A)

- Sind diese Histories Konflikt-äquivalent?

Konflikt-Äquivalenz: Konfliktrelation (2)

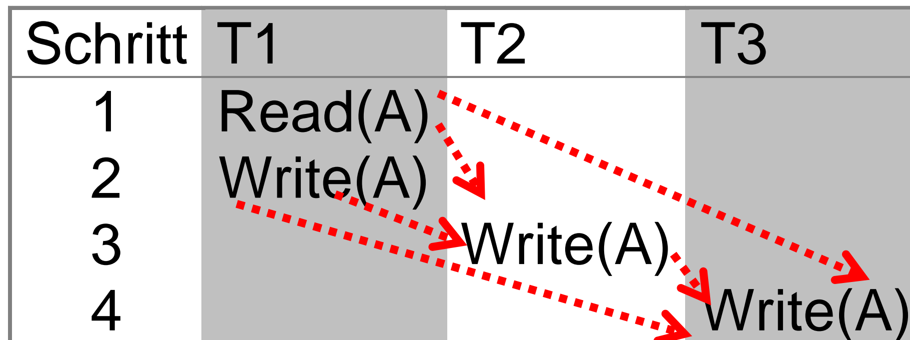
- History 1:



Schritt	T1	T2	T3
1	Read(A)		
2		Write(A)	
3	Write(A)		
4			Write(A)

Alle Transaktionen werden committen. (Unten ebenso.)

- History 2:



Schritt	T1	T2	T3
1	Read(A)		
2	Write(A)		
3		Write(A)	
4			Write(A)

- Sind diese Histories Konflikt-äquivalent?

Zusammenhang Serialisierbarkeit, History und Konflikt-Äquivalenz

- Bisher:
  - Serialisierbarkeit – Unterscheidung zwischen ‘guten’ und ‘schlechten’ Histories.
  - History kann serialisierbar sein, Dirty Reads sind jedoch trotzdem möglich.
  - Äquivalenzbegriff: Konflikt-Äquivalenz
- Im Folgenden:
  Wie bestimmt man Serialisierbarkeit einer History H auf Basis der Konflikt-Äquivalenz?

Z

Serialisierbarkeitsgraph (1)

Karlsruher Institut für Technologie

- Test, ob ein Schedule
  (= Transaktionen, zusammen mit ihren Operationen/History)
  serialisierbar ist:
- Erzeuge Serialisierbarkeitsgraphen
  bzw. **Abhängigkeitsgraphen.**
- Knoten = Transaktionen,
- (gerichtete) Kante von T_1 nach T_2 wenn Tupel (op_1[..],op_2[..]) in Konflikrelation
  - Mit op[] := r[] oder w[], mindestens ein op[] ist w[]
  - Führen damit (Teil) der äquivalenten seriellen Ordnung ein: T_1 vor T_2
  - op_1[..] zeitlich vor op_2[..]
  - Jeweils maximal eine Kante zwischen zwei Knoten

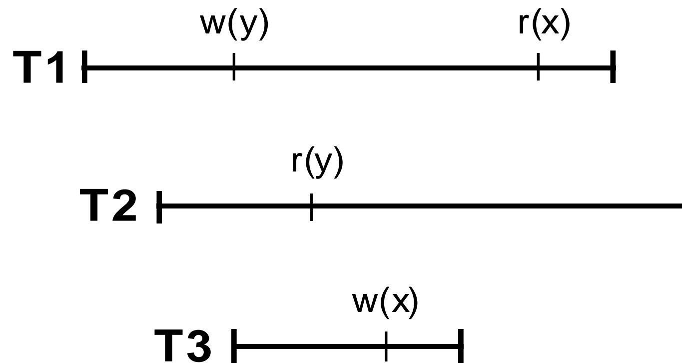
Serialisierbarkeitsgraph (3)

Karlsruher Institut für Technologie

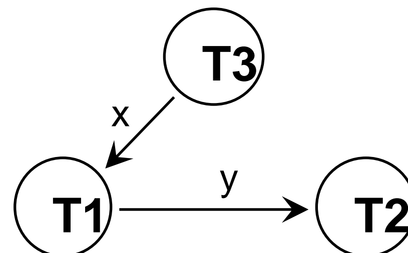
- Theorem: Schedule ist serialisierbar, wenn entsprechender Abhängigkeitsgraph *zykelfrei* ist.
- Denn: Partielle Ordnung, zu totaler Ordnung erweiterbar – äquivalenter serieller Schedule.

Serialisierbarkeitsgraph – Beispiel

- r(x)/w(x) – Lese-/Schreibzugriff auf Datenobjekt x.
- Schedule:



- Abhängigkeitsgraph:



- azyklisch → Schedule ist serialisierbar.
- Serialisierungsreihenfolge: T3 < T1 < T2

Einleitung/ Probleme

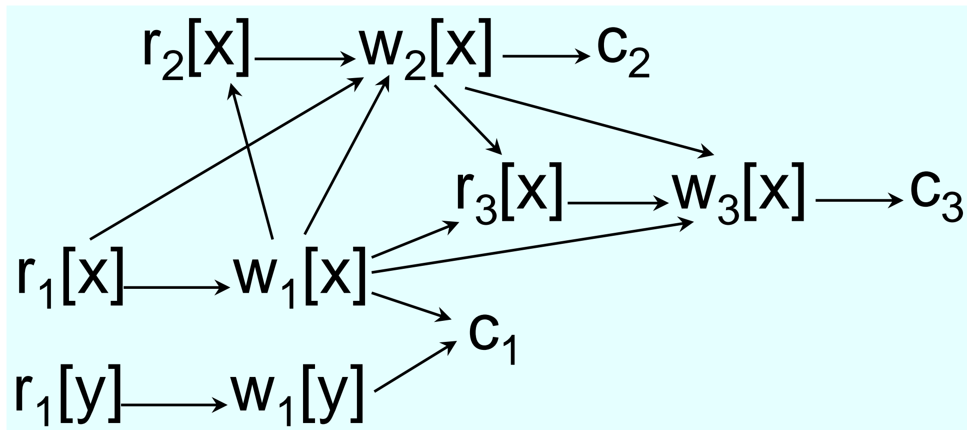
Definitionen

Locking

Dienste

Serialisierbarkeitsgraph: Bespiel

- Wie sieht Serialisierbarkeitsgraph aus?



Einleitung/ Probleme

Definitionen

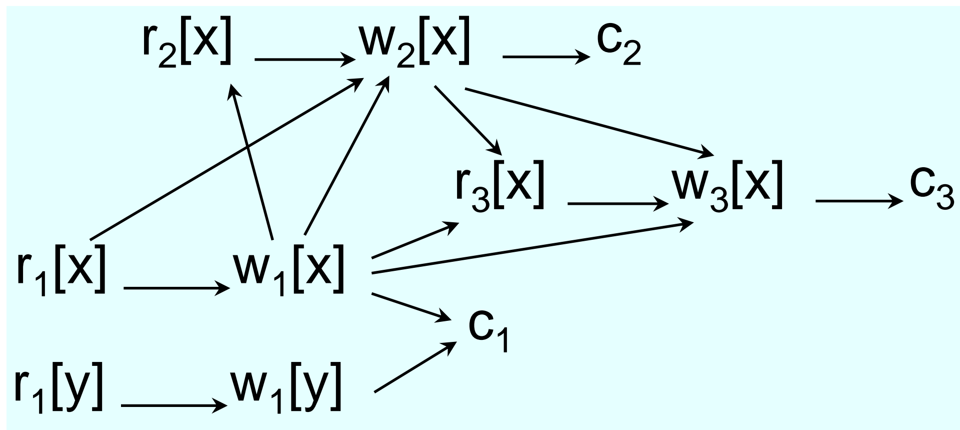
Locking

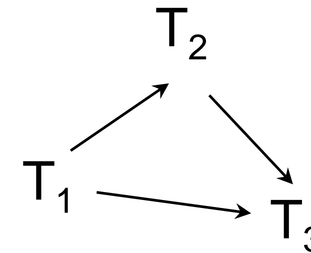
Dienste

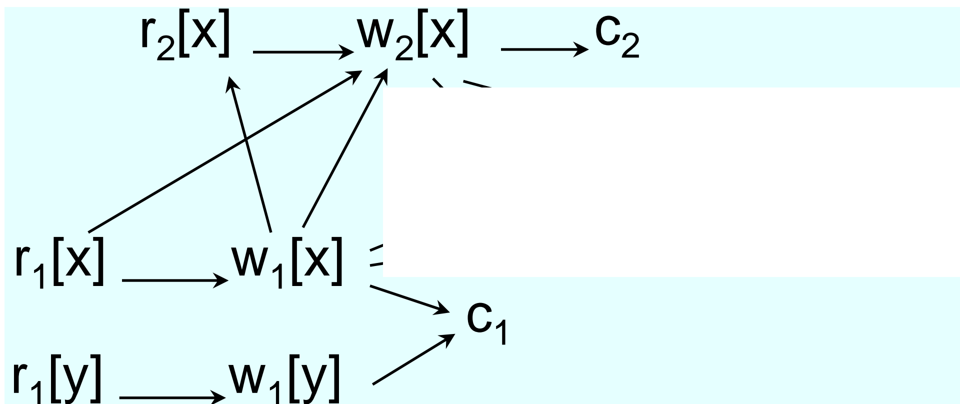
Serialisierbarkeit: Abgeschlossenheit

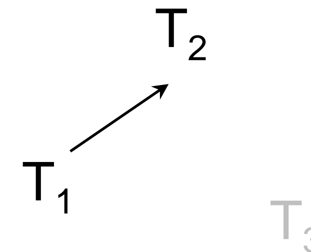
Karlsruher Institut für Technologie

- Konflikt-Serialisierbarkeit
  ist offensichtlich **prefix commit-closed**.
- Illustration:









Einleitung/
Probleme
Definitionen
Locking
Dienste

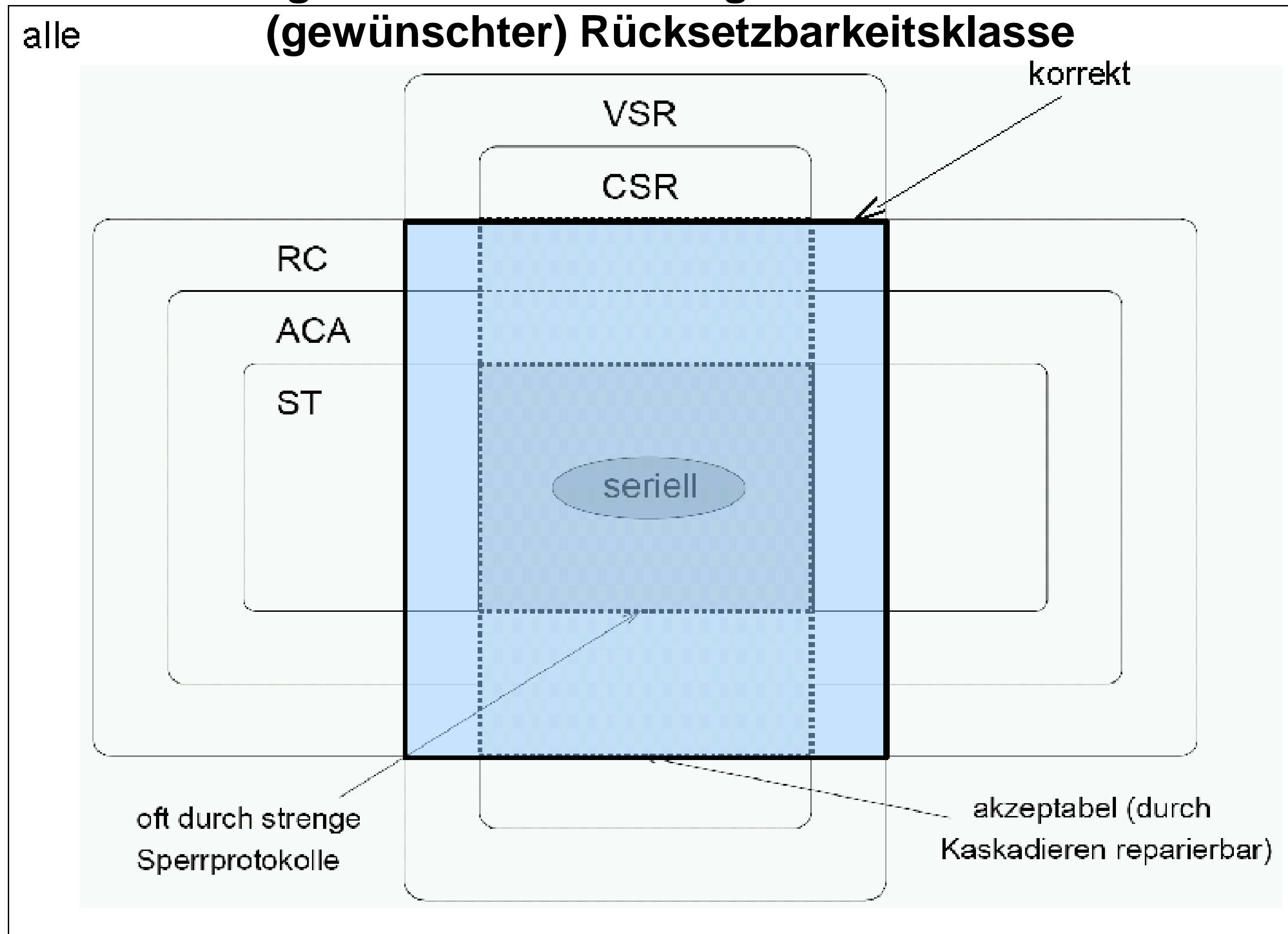
Serialisierbarkeitsgraph – Diskussion

Karlsruher Institut für Technologie

Ansatz ist nicht praktikabel.

- Serialisierbarkeit von Schedules nur im nachhinein überprüfbar
  - ➢ Zyklus kann sich erst spät ergeben, während Präfixe alle korrekt waren
- **Problem Fehlertoleranz**
  - Transaktionen können aus verschiedensten Gründen abgebrochen werden
  - H = $r_1[x] \rightarrow w_1[x] \rightarrow \mathbf{r_2[x]} \rightarrow a_1 \rightarrow w_2[x] \rightarrow c_2$
  - H ist konfliktserialisierbar, aber nicht fehlertolerant
  - Abbruch von T_1 für zu kaskadiertem Abbruch von T_2
    - Sendet user-level commit, System erkennt Problem und bricht ab (system-level abort)

Fehlertoleranz vs. Serialisierbarkeit



Rücksetzbarkeitsklassen

Ziel: DBS soll sich so verhalten, dass

- es alle Effekte von committeten Transaktionen enthält,
- keine Effekte von nicht committeten Transaktionen

Wichtige formale Definition: Liest-von-Relation:

- T_i liest x von T_j genau dann wenn T_j x vorher geschrieben hat und nicht abgebrochen wurde
- $w_j[x] < r_i[x]$
- $a_j \not< r_i[x]$

Rücksetzbar (RC)

- Ein Abort darf die Semantik einer committeten Transaktion nicht verändern!

- Soll nicht gehen: $w_1[x] \rightarrow r_2[x] \rightarrow w_2[y] \rightarrow c_2 \rightarrow a_1$
- T_2 nicht rücksetzbar, weil commited
  - ➢ widerspricht Persistenzforderung aus ACID

- Lösung: Commit für T_i erst dann erlauben, wenn alle Transaktionen T_j , von denen T_i liest, committed haben:
  - $w_1[x] \rightarrow r_2[x] \rightarrow w_2[y] \rightarrow c_1 \rightarrow c_2$

- Formale Definition RC:
  - $\forall (T_i, T_j)$ $(i \neq j)$ für die gilt T_i liest x von T_j und c_i in H: $c_j < c_i$

Vermeiden kaskadierender Abbrüche (ACA)

Karlsruher Institut für Technologie

- Rücksetzbarkeit kann immer noch zu Problemen führen:
  - Bsp: $w_1[x] \rightarrow r_2[x] \rightarrow w_2[y] \rightarrow a_1$
  - Kaskadierender Abbruch von T_2
- Lösung: Nur lesen von bereits committeten Transaktionen:
  - $w_1[x] \rightarrow c_1 \rightarrow r_2[x] \rightarrow w_2[y] \rightarrow c_2$
- Formale Definition RC:
  - $\forall\ (T_i, T_j)\ \ (i \neq j)$ für die gilt T_i liest x von T_j und c_i in H: $c_j < r_i[x]$
- Englisch: Avoid cascading aborts (ACA)

Striktheit (ST)

- "Rückgängig machen" soll einfach implementierbar sein.
- Ansatz: Before Images
  - Bsp: $w_1[x,1] \rightarrow w_1[y,3] \rightarrow \underbrace{w_2[y,1]}_{\text{BI: } y=3} \rightarrow c_1 \rightarrow r_2[x] \rightarrow a_2$
- Problem: Was tun wenn Before Image auch auf abgebrochener Transaktion basiert?
  - $x=1 \rightarrow w_1[x,2] \rightarrow w_2[x,3] \rightarrow a_1 \rightarrow a_2$
  - Lösung: Auch beim Schreiben von x darauf warten, dass alle Transaktionen die x verändert haben committed oder aborted sind.
- Formale Definition ST: Wenn $w_j(x) < o_i(x)$ $(i \neq j)$:
  - Entweder $a_j < o_i(x)$ oder $c_j < o_i(x)$
  - Es darf kein 'Objekt' x einer noch nicht beendeten Transaktion gelesen oder überschrieben werden

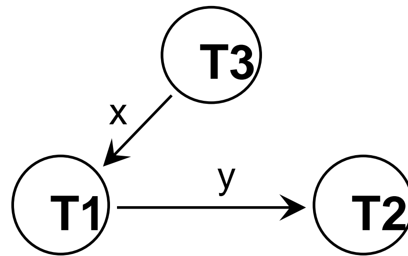
Karlsruher Institut für Technologie

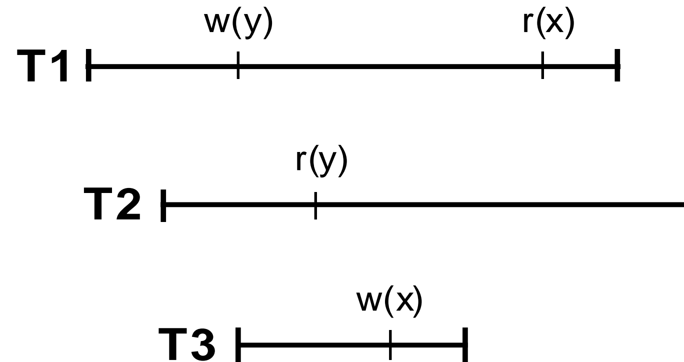
Techniken zum Erreichen der gewünschten Korrektheit

Begriffe vom letzten mal

Karlsruher Institut für Technologie

- Korrektheit := Serialisierbar und Rücksetzbarkeitsklasse
- Serialisierbarkeit
  - Konflikserialisierbarkeit (CSR)





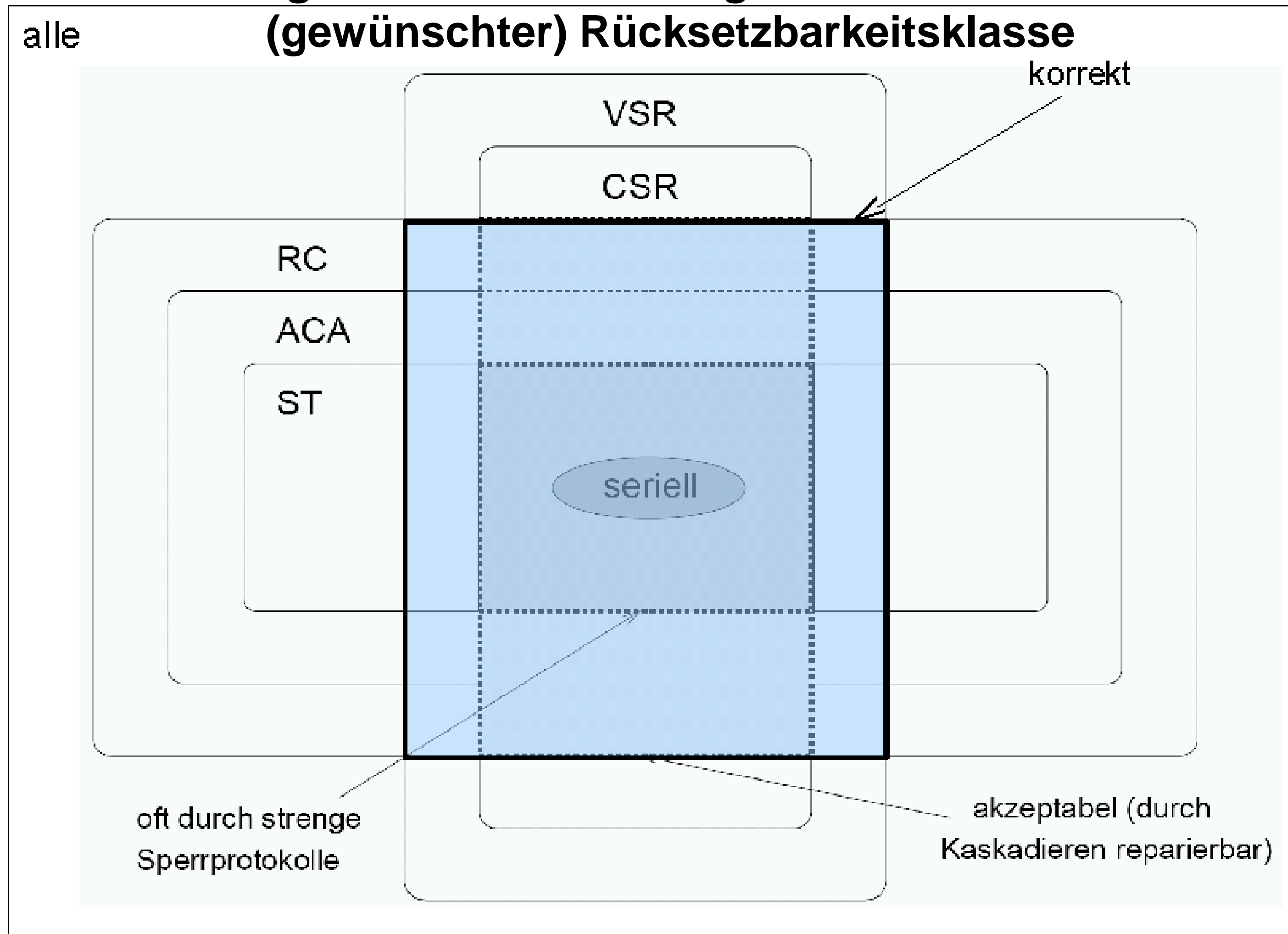
- Rücksetzbarkeitsklasse
  - Vermeiden kaskadierender Abbrüche (ACA)
  - Bsp: $w_1[x] \rightarrow r_2[x] \rightarrow w_2[y] \rightarrow a_1$
  - Kaskadierender Abbruch von T_2

- **Heute: Korrektheit mittels Locking**

Einleitung/ Probleme
Definitionen
Locking
Dienste

Fehlertoleranz vs. Serialisierbarkeit

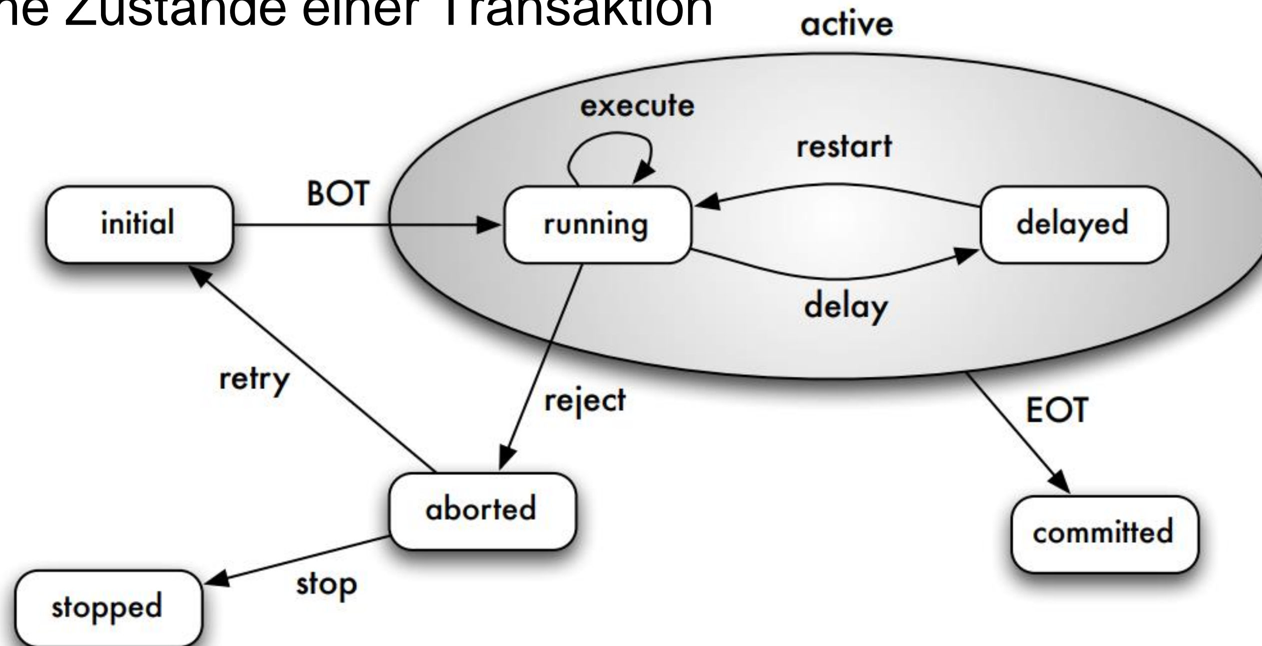
Erweiterung des Korrektheitsbegriffs: Serialisierbar und in (gewünschter) Rücksetzbarkeitsklasse



Verzögern und Zurücksetzen

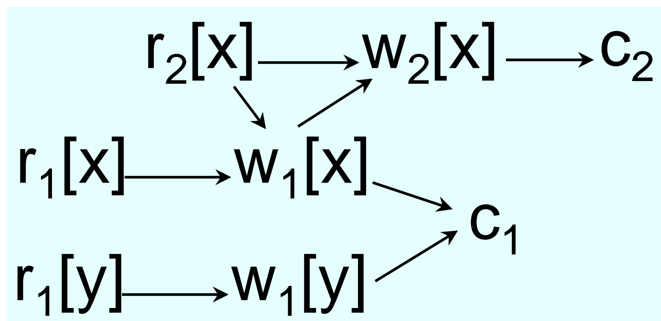
- Verzögern und Zurücksetzen sind übliche Techniken in der Transaktionsverwaltung.

- `Rollback` – Rückgängigmachen einer Transaktion
- `Restart` – Meist in Verbindung mit `rollback & restart`
- Mögliche Zustände einer Transaktion



Locking (1)

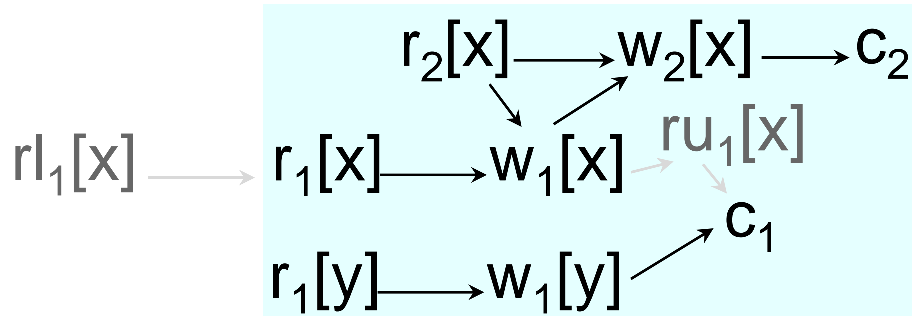
- Lock für jedes Datenobjekt und für jede Art von Operation
  - rl[x]: read lock auf Objekt x
  - wl[x]: write lock auf Objekt x
- Lock erfordert zwingend unlock
  - ru[x] and wu[x],
  - Oft zusammengefasst als unlock u[x]
- Illustration: Ohne locks



Einleitung/ Probleme
Definitionen
Locking
Dienste

Locking (2)

- Lock für jedes Datenobjekt
  und für jede Art von Operation
- Transaktion verschafft sich Lock, Transaktion gibt Lock frei.
- Illustration: Mit Beispiel-Locks



Einleitung/
Probleme
Definitionen
Locking
Dienste

Sperrdisziplin – Locking rules

- Schreibzugriff w[x] nur nach Setzen des Schreiblocks wl[x] durch entsprechende Transaktion
- Lesezugriff nach erteiltem rl[x] **oder** wl[x]
- Nur Sperren von Objekten, die nicht bereits gesperrt sind
- Nur Verschärfung von rl[x] durch wl[x] möglich
- Nach u(x) durch T_i , darf T_i x nicht mehr sperren
- Vor dem commit müssen alle Sperren aufgehoben werden

Einleitung/ Probleme
Definitionen
Locking
Dienste

Sperrdisziplin allein reicht nicht aus

Karlsruher Institut für Technologie

- Dead- und Livelocks werden nicht verhindert

Deadlock

	t_1	t_2	
	$wl(x)$		
		$wl(y)$	
delay $\rightarrow$	$wl(y)$		
	$\vdots$	$\vdots$	
		$wl(x)$	$\leftarrow$ **delay**
	Deadlock!		

Livelock

- Transaktion T_2 kommt nicht dran

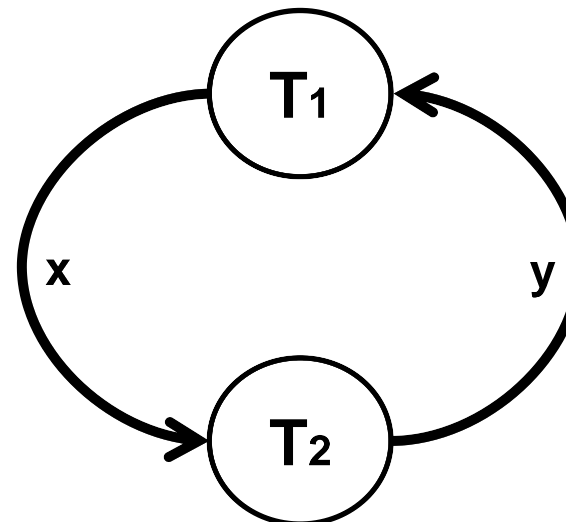
1. T_1 locks A
2. T_2 versucht lock A, muss warten
3. T_3 versucht A, muss auch warten
4. T_1 unlock A
5. T_3 ist vor T_2 dran und locked A
6. T_2 versucht lock A, muss warten
7. T_4 versucht lock A, muss warten
8. T_3 unlock A
9. T_4 ist vor T_2 dran ...

Sperrdisziplin – Allein reicht nicht aus

- Sichert auch keine Konfliktserialisierbarkeit zu

T_1	T_2
$wl(x)$	
$w(x)$	
$u(x)$	
	$wl(x)$
	$w(x)$
	$u(x)$
	$wl(y)$
	$w(y)$
	$u(y)$
$wl(y)$	
$w(y)$	
$u(y)$	

Konfliktdefinition von Locks analog zu Konflikt der Operatoren



Konflikt – Gilt analog für Locks

Karlsruher Institut für Technologie

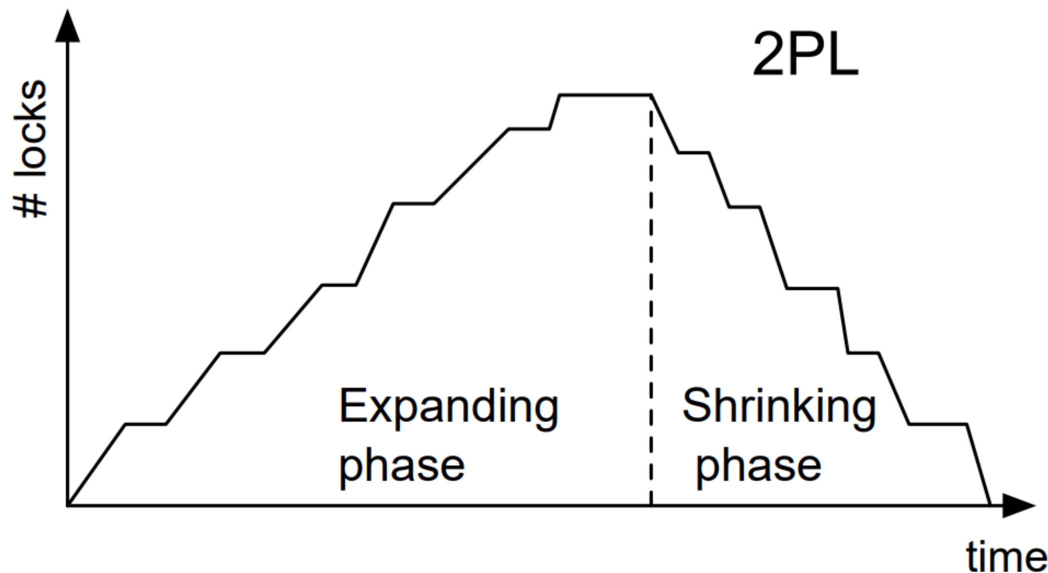
– Folie von vorhin –

- *Zwei Operationen p, q konfligieren* :=
  p, q greifen auf das gleiche Datenobjekt zu,
  und p oder q ist eine Schreiboperation.
- Weitere Operationen –
  Definition von 'Konflikt' muss erweitert werden.
- Beispiel. Kompatibilitätsmatrix:

	Read	Write
Read	**y**	**n**
Write	**n**	**n**

Locking (4)

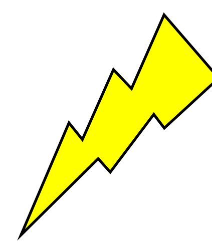
- Da Locking für sich betrachtet ist nicht ausreichend: Zwei Phasen:
  - 1. Phase, in der Locks hinzugenommen werden,
  - 2. Phase, in der Locks freigegeben werden.
- **Zwei-Phasen Sperrprotokoll** (two-phase locking, 2PL) stellt Serialisierbarkeit sicher.



Einleitung/ Probleme
Definitionen
Locking
Dienste

Beispiel für Deadlock mit 2PL

- T_1: $r_1[x] \rightarrow w_1[y] \rightarrow c_1$; T_2: $w_2[y] \rightarrow w_2[x] \rightarrow c_2$
- Abfolge:
  1. Beide Transaktionen halten zunächst keine Locks.
  2. TM sendet $r_1[x]$ an Scheduler. $rl_1[x]$, Scheduler sendet $r_1[x]$ an DM.
  3. TM sendet $w_2[y]$ an Scheduler. $wl_2[y]$, Scheduler sendet $w_2[y]$ an DM.
  4. TM sendet $w_2[x]$ an Scheduler. $wl_2[x]$ nicht möglich. Verzögerung.
  5. TM sendet $w_1[y]$ an Scheduler. $wl_1[y]$ nicht möglich. Verzögerung.

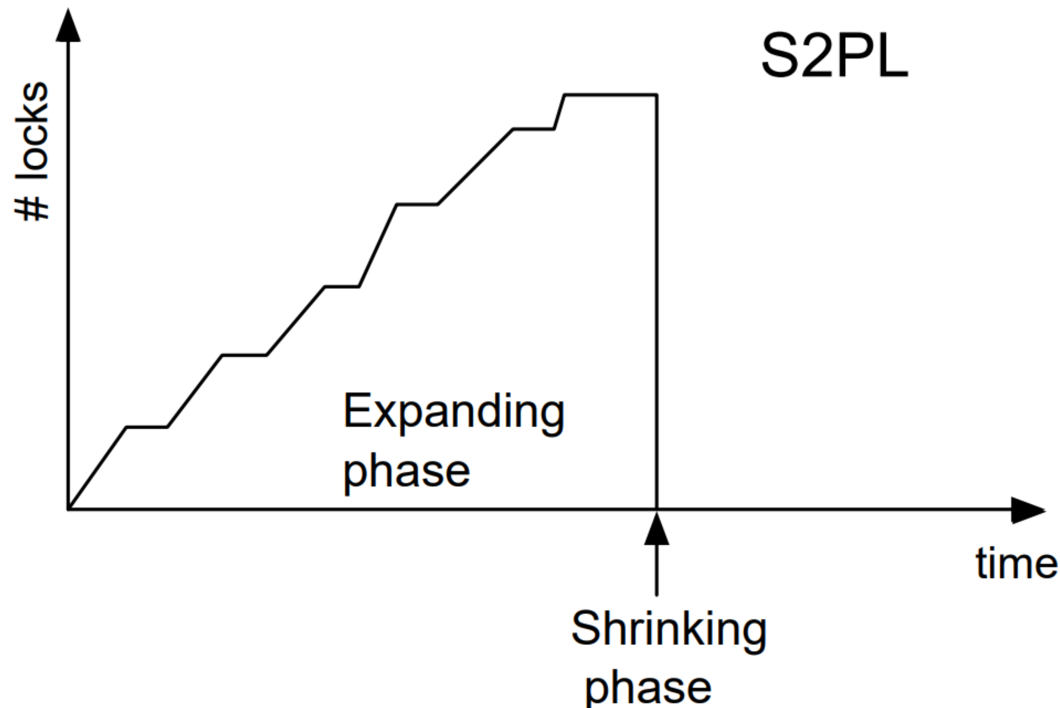


- Deadlock. Muss extern zurückgesetzt werden.

Einleitung/ Probleme
Definitionen
Locking
Dienste

Strenges Zwei-Phasen Sperrprotokoll

- **Strenges Zwei-Phasen Sperrprotokoll** (S2PL) durch atomare Freigabephase:
- Vermeidet kaskadierende Abbrüche durch Freigabe der Locks erst am Ende der Transaktion.
- Weniger Histories möglich, keine Deadlock-Vermeidung



Einleitung/ Probleme

Definitionen

Locking

Dienste

Dirty Read

– Folie von vorhin –

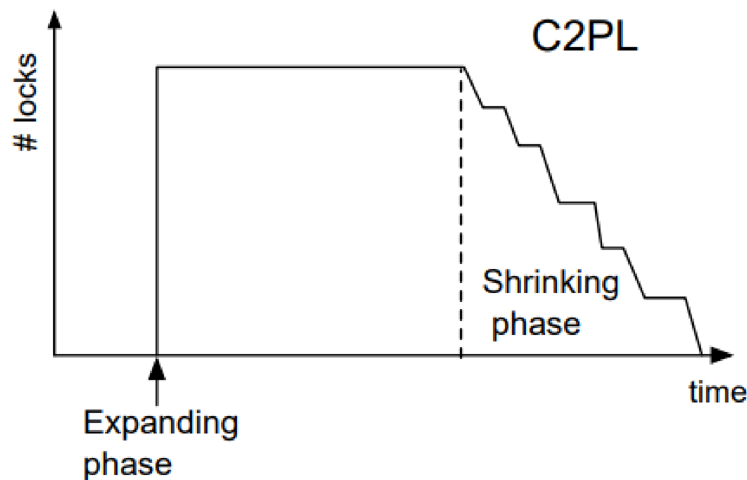
- *Commit*, *Abort*.
- Programm T_2 schreibt Zinsen gut, basierend auf einem Wert, der nicht zu einem konsistenten Zustand gehört.
- Denn später erfolgt Abort von T_1.

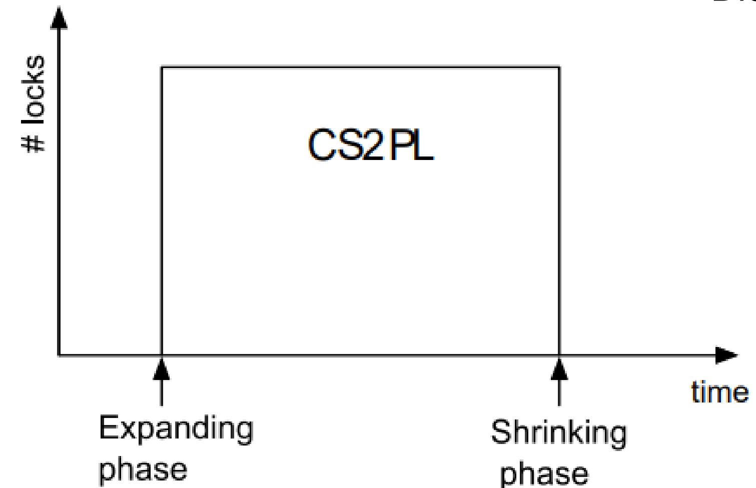
Schritt	*T1*	*T2*
1	Read(A, a1)	
2	a1 := a1-300	
3	Write(A, a1)	
4		Read(A, a2)
5		a2 := a2 *1.03
6		Write(A, a2)
7		**commit**
8	Read(B, b1)	
9	…	
10	**abort**	

Warum widerspricht das dem **Strengen** Zwei-Phasen Sperrprotokoll?

Konservatives 2PL (C2PL)

- **Konservatives Zwei-Phasen Sperrprotokoll** (S2PL) durch atomare Anforderungsphase:
- Vermeidet **Deadlocks**
- Weniger Histories möglich, kein ACA
- Operationen müssen zuvor bekannt sein





Zusammenfassung

Karlsruher Institut für Technologie

- Nebenläufiger Zugriff – fundamental wichtiges Anliegen.
- Serielle Ausführung wäre korrekt,
  ist wegen mangelhafter Performance aber nicht akzeptabel.
- Korrektheitskriterium – Äquivalenz zu serieller Ausführung.
- Two-Phase Locking stellt Serialisierbarkeit sicher.

Mögliche Prüfungsfragen (1)

- Was ist Isolation? Was ist der Zusammenhang zwischen Isolation und Serialisierbarkeit?
- Welche Probleme können bei unkontrollierter nebenläufiger Ausführung von Transaktionen auftreten?

Mögliche Prüfungsfragen (2)

- <Beispiele geben können für Lost Updates, Non-Repeatable Reads usw., die bestimmte Bedingungen erfüllen/die sich von denen aus der Vorlesung unterscheiden.>
- Warum ist es wichtig, dass unser Korrektheitskriterium für Histories prefix commit closed ist?
  Erklären Sie, warum Konflikt-Serialisierbarkeit prefix commit closed ist.

Mögliche Prüfungsfragen (3)

Karlsruher Institut für Technologie

- Ist eine gegebene History serialisierbar/recoverable/cascadeless?
- Haben zwei Konflikt-äquivalente Histories
  stets die gleichen Reads-from Beziehungen?
- Warum verwendet man i. d. R. nicht den Serialisierbarkeitsgraphen, um Serialisierbarkeit sicherzustellen?
- Bei Deadlocks wird i. d. R. eine Transaktion zurückgesetzt. Kann es vorkommen, dass die gleiche Transaktion (a) mehrmals (b) beliebig oft zurückgesetzt wird? Wenn ja, was kann man jeweils dagegen tun?

Mögliche Prüfungsfragen (4)

Karlsruher Institut für Technologie

- Geben Sie ein Beispiel für eine serialisierbare Ausführung bestehend aus drei Transaktionen mit folgender Eigenschaft: Die zeitliche Reihenfolge der Commits ist c1 vor c2 vor c3, die der äquivalenten seriellen Ausführung jedoch c3 vor c2 vor c1.
- Um ein Deadlock aufzulösen, muß eine der beteiligten Transaktionen zurückgesetzt werden. Welche Kriterien sind Ihres Erachtens sinnvoll, um diese Auswahl zu treffen?