LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Skript zur Vorlesung:

Datenbanksysteme II

Sommersemester 2013

Kapitel 3

Synchronisation

Vorlesung: PD Dr. Peer Kröger

http://www.dbs.ifi.lmu.de/cms/Datenbanksysteme_II

Dieses Skript basiert im Wesentlichen auf den Skripten zur Vorlesung Datenbanksysteme II an der LMU München von

Prof. Dr. Christian Böhm (SoSe 2007),
PD Dr. Peer Kröger (SoSe 2008) und
PD Dr. Matthias Schubert (SoSe 2009)

3. Synchronisation

3. Synchronisation

3.1 Einleitung

Synchronisation (Concurrency Control)

- Serielle Ausführung von Transaktionen (=***Isolation***)
  - unerwünscht, da die Leistungsfähigkeit des Systems beeinträchtigt ist
  - Folgen: niedriger Durchsatz, hohe Wartezeiten
- Mehrbenutzerbetrieb
  - führt i.A. zu einer besseren Auslastung des Systems (z.B. Wartezeiten bei E/A-Vorgängen können zur Bearbeitung anderer Transaktionen genutzt werden)
  - Aufgabe der Synchronisation
  - Gewährleistung des logischen Einbenutzerbetriebs, d.h. innerhalb einer TA ist ein Benutzer von den Aktivitäten anderer Benutzer nicht betroffen

3.1 Einleitung

Anomalien im Mehrbenutzerbetrieb

- Klassifikation
  - Verloren gegangene Änderungen (Lost Updates)
  - Zugriff auf „schmutzige“ Daten (Dirty Read / Dirty Write)
  - Nicht-reproduzierbares Lesen (Non-Repeatable Read)
  - Phantomproblem

- Beispiel: Flugdatenbank

Passagiere	FlugNr	Name	Platz	Gepäck
	LH745	Müller	3A	8
	LH745	Meier	6D	12
	LH745	Huber	5C	14
	BA932	Schmidt	9F	9
	BA932	Huber	5C	14

3.1 Einleitung

Lost Updates

- Änderungen einer TA können durch Änderungen anderer TA überschrieben werden und dadurch verloren gehen

- Bsp.: Zwei Transaktionen T1 und T2 führen je eine Änderung auf demselben Objekt aus

```
T1:   UPDATE Passagiere SET Gepäck = Gepäck+3
            WHERE FlugNr = LH745 AND Name = „Meier“;
T2:   UPDATE Passagiere SET Gepäck = Gepäck+5
            WHERE FlugNr = LH745 AND Name = „Meier“;
```

3.1 Einleitung

- Möglicher Ablauf

T1	T2

`read(Passagiere.Gepäck, x1);`
 	
`x1 := x1+3;`
`write(Passagiere.Gepäck, x1);`
	`read(Passagiere.Gepäck, x2);`
	`x2 := x2 + 5;`
	`write(Passagiere.Gepäck, x2);`

- In der DB ist nur die Änderung von T1 wirksam, die Änderung von T2 ist verloren gegangen

  → Verstoß gegen ***Durability***

3.1 Einleitung

Dirty Read / Dirty Write

- Zugriff auf „schmutzige“ Daten, d.h. auf Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden

- Beispiel:
  - T1 erhöht das Gepäck um 3 kg, wird aber später abgebrochen
  - T2 erhöht das Gepäck um 5 kg und wird erfolgreich abgeschlossen

3.1 Einleitung

- Möglicher Ablauf:

T1	T2

`UPDATE Passagiere`
`SET Gepäck = Gepäck+3;`
 	
`ROLLBACK;`
	
	`UPDATE Passagiere`
	`SET Gepäck = Gepäck+5;`
	`COMMIT;`

- Durch Abbruch von T1 werden die geänderten Werte ungültig, die T2 gelesen hat (Dirty Read). T2 setzt weitere Änderungen darauf auf (Dirty Write)

→ Verstoß gegen

- ***Consistency***: Ablauf verursacht inkonsistenten DB-Zustand

oder

- ***Durability***: T2 muss zurückgesetzt werden

3.1 Einleitung

Non-Repeatable Read

- Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts

- Beispiel:
  - T1 liest das Gepäckgewicht der Passagiere auf Flug BA932 zwei mal
  - T2 bucht den Platz 3F auf dem Flug BA932 für Passagier Meier mit 5kg Gepäck

3.1 Einleitung

- Möglicher Ablauf:

T1	T2
`SELECT Gepäck FROM Passagiere` `WHERE FlugNr = „BA932“;`	

	`INSERT INTO Passagiere`
	`VALUES (BA932, Meier, 3F, 5);`
	`COMMIT;`

`SELECT Gepäck FROM Passagiere` `WHERE FlugNr = „BA932“;`	

- Die beiden SELECT-Anweisungen von Transaktion T1 liefern unterschiedliche Ergebnisse, obwohl T1 den DB-Zustand nicht geändert hat

→ Verstoß gegen ***Isolation***

3.1 Einleitung

Phantomproblem

- Ausprägung (Spezialfall) des nicht-reproduzierbaren Lesens, bei der neu generierte Daten, sowie meist bei der 2. TA Aggregat-Funktionen beteiligt sind

- Bsp.:
  - T1 druckt die Passagierliste sowie die Anzahl der Passagiere für den Flug LH745
  - T2 bucht den Platz 7D auf dem Flug LH745 für Phantomas

3.1 Einleitung

- Möglicher Ablauf

T1	T2

`SELECT * FROM Passagiere`
`WHERE FlugNr = „LH745“;`
 	
`SELECT COUNT(*) FROM Passagiere`
`WHERE FlugNr = „ LH745“;`
	`INSERT INTO Passagiere`
	`VALUES (LH745, Phantomas, 7D, 2);`
	`COMMIT;`

- Für Transaktion T1 erscheint Phantomas noch nicht auf der Passagierliste, obwohl er in der danach ausgegebenen Anzahl der Passagiere berücksichtigt ist

3. Synchronisation

3.2 Serialisierbarkeit von Transaktionen

Motivation

- Bearbeitung von Transaktionen
  - Nebenläufigkeit vor den Benutzern verbergen
  - Transparent für den Benutzer, als ob

    TAs (in einer beliebigen Reihenfolge) hintereinander ausgeführt werden

    und NICHT als ob

    TAs ineinander verzahnt ablaufen und sich dadurch (unbeabsichtigt) beeinflussen

3.2 Serialisierbarkeit von Transaktionen

Schedules

- Allgemeiner Schedule:
  Ein Schedule („Historie“) für eine Menge {T1, …, T_n} von Transaktionen ist eine Folge von Aktionen, die durch Mischen der Aktionen der Transaktionen T_i entsteht, wobei die Reihenfolge innerhalb der jeweiligen Transaktion beibehalten wird.

- Allgemeine Schedules bieten offenbar eine beliebige Verzahnung und sind daher aus Performanz-Gründen erwünscht

- Frage: Warum darf die Reihenfolge der Aktionen innerhalb einer TA nicht verändert werden?

3.2 Serialisierbarkeit von Transaktionen

- Serieller Schedule (=Isolation):
  Ein serieller Schedule ist ein Schedule S von $\{T_1, \ldots, T_n\}$, in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt sondern in Blöcken hintereinander ausgeführt werden.
- Aus Sicht des Isolation-Prinzips werden serielle Schedules benötigt

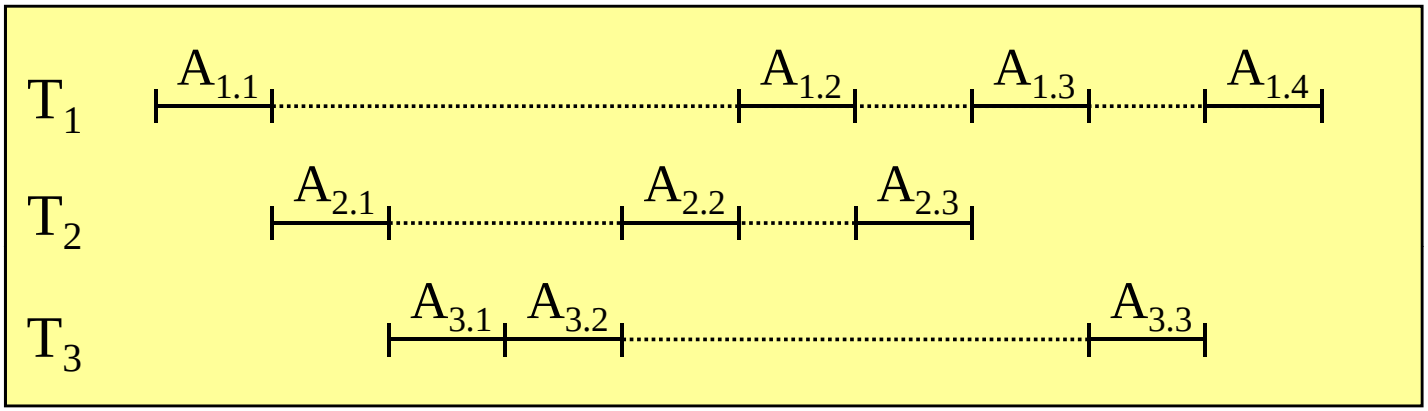
Kompromiss:

- Serialisierbarer Schedule:
  Ein (allgemeiner) Schedule S von $\{T_1, \ldots, T_n\}$ ist serialisierbar, wenn er dieselbe Wirkung hat wie ein beliebiger serieller Schedule von $\{T_1, \ldots, T_n\}$.

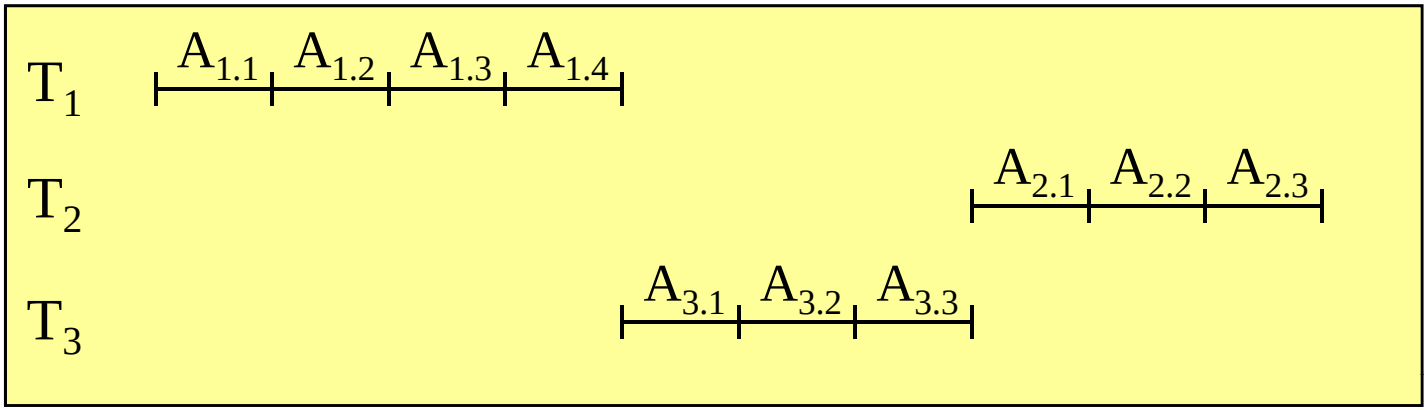
- ***Nur serialisierbare Schedules dürfen zugelassen werden!***

3.2 Serialisierbarkeit von Transaktionen

- Beispiele
  - Beliebiger Schedule:



  - Serieller Schedule:



Wirkung von Schedules

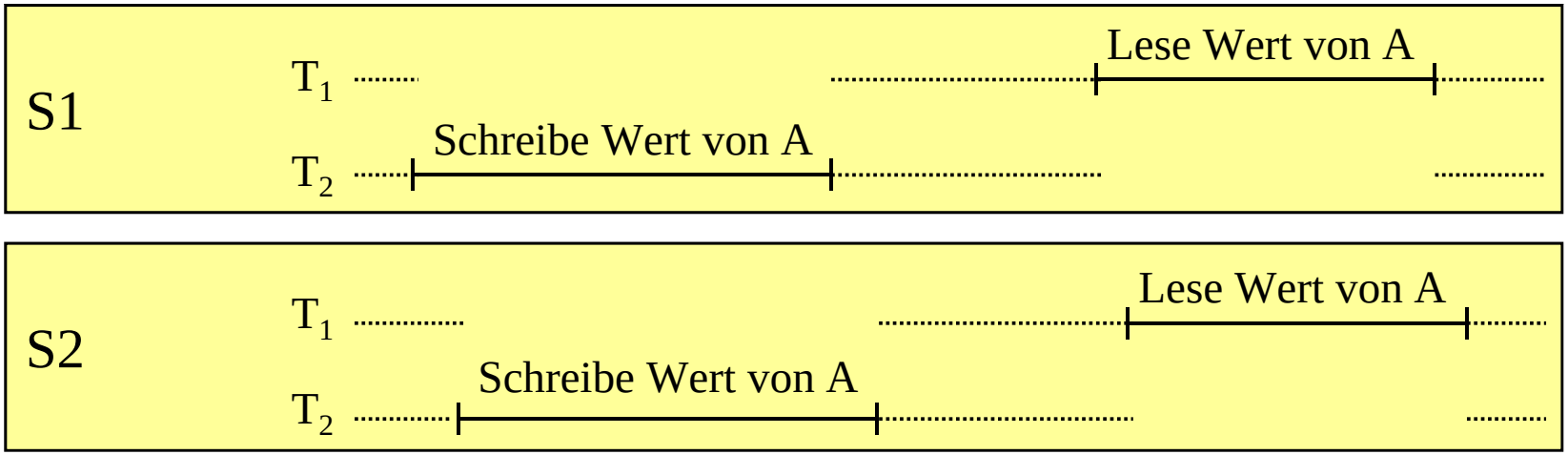
- Frage: Wann haben zwei Schedules S1 und S2 die gleiche Wirkung auf den Datenbank-Inhalt?
- Achtung:
  - Gleiches Ergebnis kann u.a. Ergebnis eines Zufalls sein
  - Dies könnte aber nur durch nachträgliches Überprüfen der Datenbank-Zustände nach S1 und S2 festgestellt werden.

3.2 Serialisierbarkeit von Transaktionen

- Wir benötigen ein objektivierbares Kriterium:

  Konflikt-Äquivalenz

- Idee: Wenn in S1 eine Transaktion T_1 z.B. einen Wert liest, den T_2 geschrieben hat, dann muss das auch in S2 so sein.



- Wir sprechen hier von einer Schreib-Lese-Abhängigkeit (bzw. Konflikt) zwischen T_2 und T_1 (in Schedule S1 und S2)

Abhängigkeiten

Sei S ein Schedule. Wir sprechen von einer

- Schreib-Lese-Abhängigkeit von $T_i \rightarrow T_j$
  - Es existiert Objekt x, so dass in S $w_i(x)$ vor $r_j(x)$ kommt
  - Abkürzung: $wr_{i,j}(x)$
- Lese-Schreib-Abhängigkeit von $Ti \rightarrow Tj$
  - Es existiert Objekt x, so dass in S $r_i(x)$ vor $w_j(x)$ kommt
  - Abkürzung: $rw_{i,j}(x)$
- Schreib-Schreib-Abhängigkeit von $T_i \rightarrow T_j$
  - Es existiert Objekt x, so dass in S $w_i(x)$ vor $w_j(x)$ kommt
  - Abkürzung: $ww_{i,j}(x)$
- Warum keine Lese-Lese-Abhängigkeiten?

Konfliktäquivalenz von Schedules

- Zwei Schedules S1 und S2 heißen konfliktäquivalent, wenn
  - S1 und S2 die gleichen Transaktions- und Aktionsmengen besitzen, d.h. wenn beide Schedules dieselben Operationen ausführen.
  - S1 und S2 die gleichen Abhängigkeitsmengen besitzen, d.h. wenn in der Abhängigkeitsmenge von S1 z.B. die Schreib-Lese-Abhängigkeit "$w_i(x)$ vor $r_j(x)$" vorkommt (für ein Objekt x), dann muss diese auch in der Abhängigkeitsmenge von S2 vorkommen.
- Zwei konflikt-äquivalente Schedules haben die gleiche Wirkung auf den Datenbank-Inhalt. (Gilt die Umkehrung?)

3.2 Serialisierbarkeit von Transaktionen

- Beispiel:

$S_1 = (r_1(x),\ r_1(y),\ r_2(x),\ w_2(x),\ w_1(x),\ w_1(y))$

$S_2 = (r_2(x),\ r_1(x),\ r_1(y),\ w_2(x),\ w_1(x),\ w_1(y))$

$S_3 = (r_1(x),\ r_1(y),\ r_2(x),\ w_1(x),\ w_2(x),\ w_1(y))$

$S_4 = (r_2(x),\ r_1(y),\ r_1(x),\ w_2(x),\ w_1(y),\ w_1(x))$

$r_i(x)$ = T_i liest x

$w_i(x)$ = T_i schreibt x

  - Aktionsmengen von S1, S2 und S3 sind identisch
  - Abhängigkeitsmengen:

$$A_{S1} = \{rw_{1,2}(x),\ rw_{2,1}(x),\ ww_{2,1}(x)\}$$
$$A_{S2} = \{rw_{2,1}(x),\ rw_{1,2}(x),\ ww_{2,1}(x)\}$$
$$A_{S3} = \{rw_{1,2}(x),\ rw_{2,1}(x),\ ww_{1,2}(x)\}$$

  - Schedule S1 und S2 sind konfliktäquivalent
  - Schedule S1 und S3, bzw. S2 und S3 sind nicht konfliktäquivalent
  - Schedule S4 ist kein Schedule derselben Transaktionen, da die Aktionen transaktionsintern vertauscht sind.

3.2 Serialisierbarkeit von Transaktionen

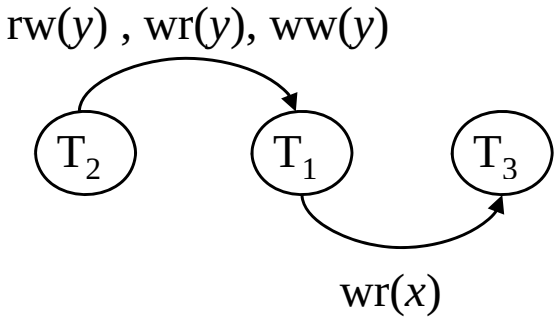
Serialisierungs-Graph

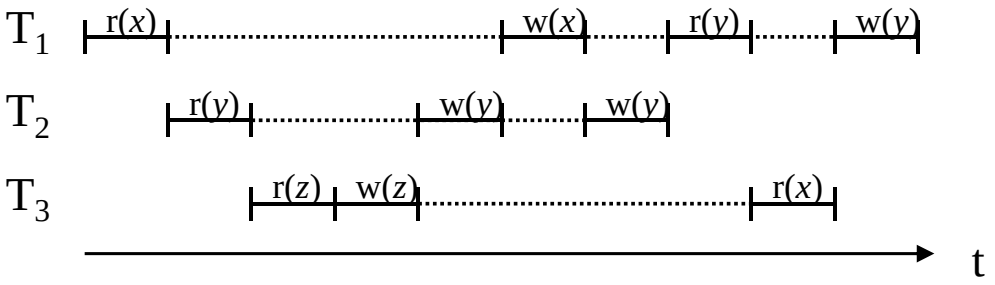
- Überprüfung, ob ein Schedule von $\{T_1, \ldots, T_n\}$ serialisierbar ist (d.h. ob ein konflikt-äquivalenter serieller Schedule existiert)
- Die beteiligten Transaktionen $\{T_1, \ldots, T_n\}$ sind die Knoten des Graphen
- Die Kanten beschreiben die Abhängigkeiten der Transaktionen:
  Eine Kante $T_i \rightarrow T_j$ wird eingetragen, falls im Schedule
  - $w_i(x)$ vor $r_j(x)$ kommt: Schreib-Lese-Abhängigkeiten $wr(x)$
  - $r_i(x)$ vor $w_j(x)$ kommt: Lese-Schreib-Abhängigkeiten $rw(x)$
  - $w_i(x)$ vor $w_j(x)$ kommt: Schreib-Schreib-Abhängigkeiten $ww(x)$

  Die Kanten werden mit der Abhängigkeit beschriftet.

3.2 Serialisierbarkeit von Transaktionen

- Es gilt:
  - Ein Schedule ist serialisierbar, falls der Serialisierungs-Graph ***zyklenfrei*** ist
  - Einen zugehörigen konfliktäquivalenten seriellen Schedule erhält man durch topologisches Sortieren des Graphen (***Serialisierungsreihenfolge***)
  - Es kann i.A. mehrere serielle Schedules geben.
  - Beispiel: $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$

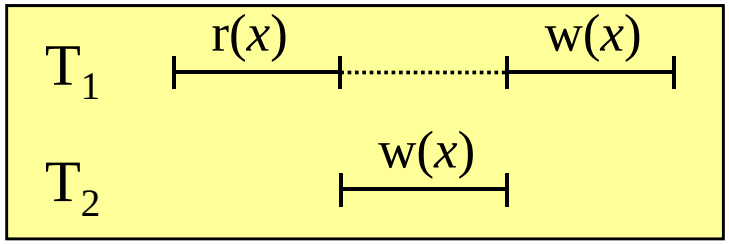


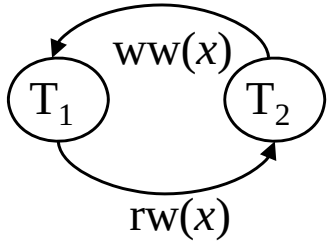


Serialisierungsreihenfolge: (T_2, T_1, T_3)

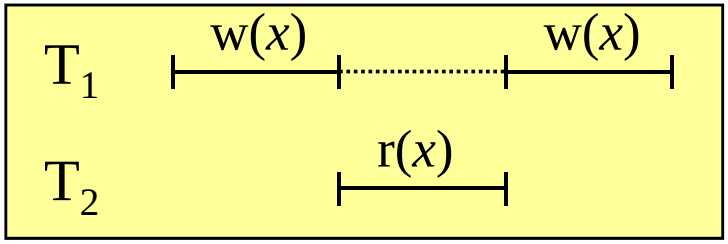
Beispiele für nicht-serialisierbare Schedules

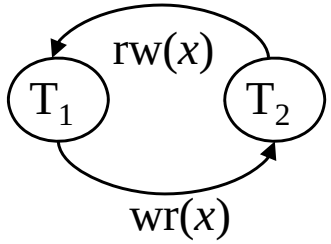
Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



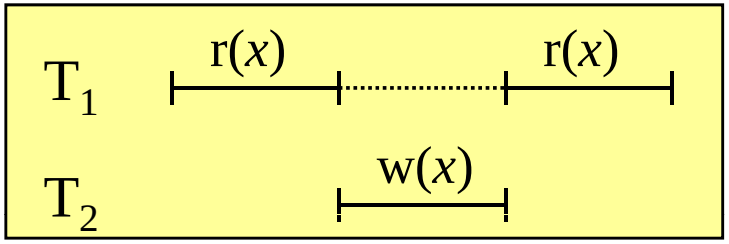


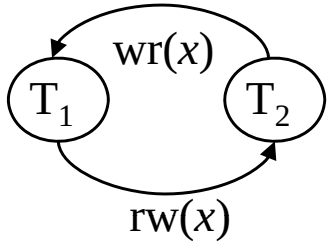
Dirty Read: $S=(w_1(x), r_2(x), w_1(x))$





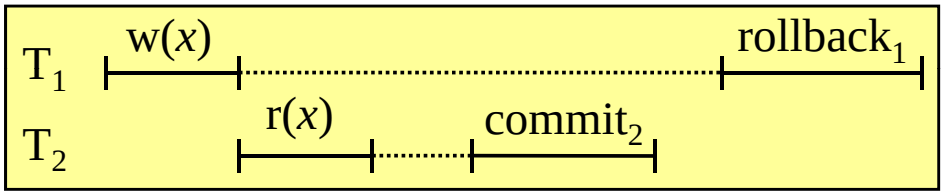
Non-repeatable Read: $S=(r_1(x), w_2(x), r_1(x))$

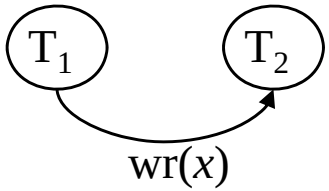




Rücksetzbare Schedules

- Bisher: Serialisierbarkeit
- Frage: was passiert, wenn eine Transaktion (z.B. auf eigenen Wunsch) zurückgesetzt wird?
- Beispiel:
  - T_1 schreibt Datensatz *x*
  - T_2 liest Datensatz *x*
  - T2 führt *COMMIT* aus
  - Schedule ist serialisierbar,
    der Serialisierungs-Graph ist zyklenfrei





  - ABER
    - T_1 wird zurückgesetzt (d.h. Datensatz *x* wird wieder auf den Ursprungswert zurückgesetzt)
    - T_2 müsste eigentlich auch zurückgesetzt werden, hat aber schon *COMMIT* ausgeführt

3.2 Serialisierbarkeit von Transaktionen

- Also: Serialisierbarkeit alleine reicht leider nicht aus, wenn TAs zurückgesetzt werden können
- ***Rücksetzbarer Schedule***:
  Eine Transaktion T_i darf erst dann ihr COMMIT durchführen, wenn alle Transaktionen T_j, von denen sie Daten gelesen hat, beendet sind.
- Andernfalls Problem: Falls ein T_j noch zurückgesetzt wird, müsste auch T_i zurückgesetzt werden, was nach *COMMIT* (T_i) nicht mehr möglich wäre

3.2 Serialisierbarkeit von Transaktionen

- Noch schlimmer:

  Rücksetzbare Schedules können eine Lawine weiterer Rollbacks in Gang setzen

Schritt	T_1	T_2	T_3	T_4	T_5
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.				$w_5(D)$	
8.					$r_5(D)$
9.	$abort_1$				

- ***Schedule ohne kaskadierendes Rücksetzen***:
  Änderungen werden erst nach dem *COMMIT* für andere Transaktionen zum Lesen freigegeben

Überblick: Scheduleklassen

- Serieller S.
  - TAs in einzelnen Blöcken, phys. Einbenutzerbetrieb
- Serialisierbarer S.
  - Konfliktäquivalent zu einem seriellen S.
- Rücksetzbarer S.
  - TA darf erst committen, wenn alle TAs von denen sie Daten gelesen hat committed haben
- S. ohne kaskadierendes Rollback
  - Veränderte Daten einer noch laufenden TA dürfen nicht gelesen werden
- Strikter S.
  - Zusätzlich dürfen veränderte Daten einer noch laufenden TA nicht überschrieben werden

3.2 Serialisierbarkeit von Transaktionen

- Überblick: Beziehungen zwischen Scheduleklassen