

Skript zur Vorlesung:  
**Datenbanksysteme**  
Wintersemester 2018/2019

# Kapitel 9a

# Transaktionen - Synchronisation

Vorlesung: Prof. Dr. Christian Böhm

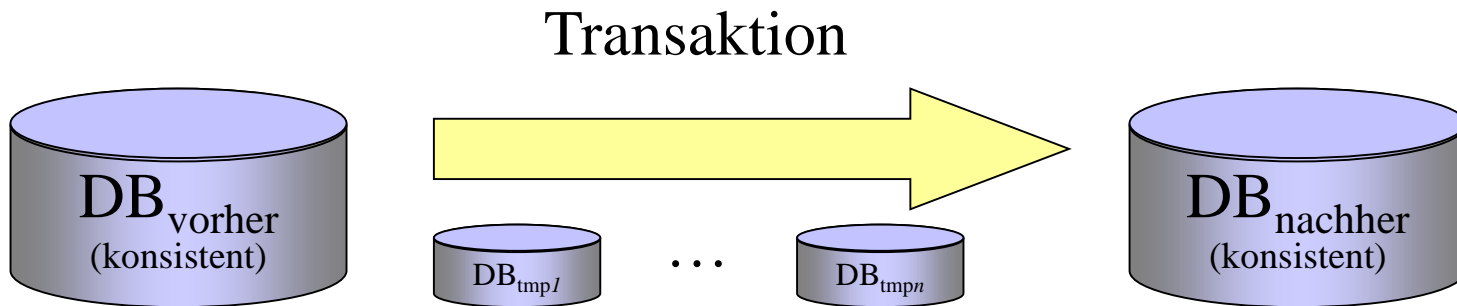
Übungen: Dominik Mautz

<http://dmm.dbs.ifi.lmu.de/dbs>



# Transaktionskonzept

- Transaktion: Folge von Befehlen (*read, write*), die die DB von einen **konsistenten** Zustand in einen anderen **konsistenten** Zustand überführt
- Transaktionen: Einheiten **integritätserhaltender Zustandsänderungen** einer Datenbank
- Hauptaufgaben der Transaktions-Verwaltung
  - Synchronisation (Koordination mehrerer Benutzerprozesse)
  - Recovery (Behebung von Fehlersituationen)

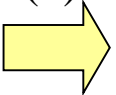
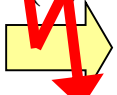


# Transaktionskonzept

Beispiel Bankwesen:

Überweisung von Huber an Meier in Höhe von 200 €

- Mgl. Bearbeitungsplan:
  - (1) Erniedrige Stand von Huber um 200 €
  - (2) Erhöhe Stand von Meier um 200 €
- Möglicher Ablauf

Konto	Kunde	Stand	(1)	Konto	Kunde	Stand	(2)
	Meier	1.000 €			Meier	1.000 €	
	Huber	1.500 €			Huber	1.300 €	

**System-absturz**

**Inkonsistenter DB-Zustand darf nicht entstehen bzw.  
darf nicht dauerhaft bestehen bleiben!**

# Eigenschaften von Transaktionen

- **ACID-Prinzip**

- **Atomicity** (Atomarität)

- Der Effekt einer Transaktion kommt entweder ganz oder gar nicht zum Tragen.

- **Consistency** (Konsistenz, Integritätserhaltung)

- Durch eine Transaktion wird ein konsistenter Datenbankzustand wieder in einen konsistenten Datenbankzustand überführt.

- **Isolation** (Isoliertheit, logischer Einbenutzerbetrieb)

- Innerhalb einer Transaktion nimmt ein Benutzer Änderungen durch andere Benutzer nicht wahr.

- **Durability** (Dauerhaftigkeit, Persistenz)

- Der Effekt einer abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten.

- Weitere Forderung: TA muss in endlicher Zeit bearbeitet werden können

# Steuerung von Transaktionen

- **begin of transaction (BOT)**
  - markiert den Anfang einer Transaktion
  - In SQL werden Transaktionen implizit begonnen, es gibt kein `begin work` o.ä.
- **end of transaction (EOT)**
  - markiert das Ende einer Transaktion
  - alle Änderungen seit dem letzten BOT werden festgeschrieben
  - SQL: `commit` oder `commit work`
- **abort**
  - markiert den Abbruch einer Transaktion
  - die Datenbasis wird in den Zustand vor BOT zurückgeführt
  - SQL: `rollback` oder `rollback work`
- **Beispiel**

```
UPDATE Konto SET Stand = Stand-200 WHERE Kunde = 'Huber';  
UPDATE Konto SET Stand = Stand+200 WHERE Kunde = 'Meier';  
COMMIT;
```

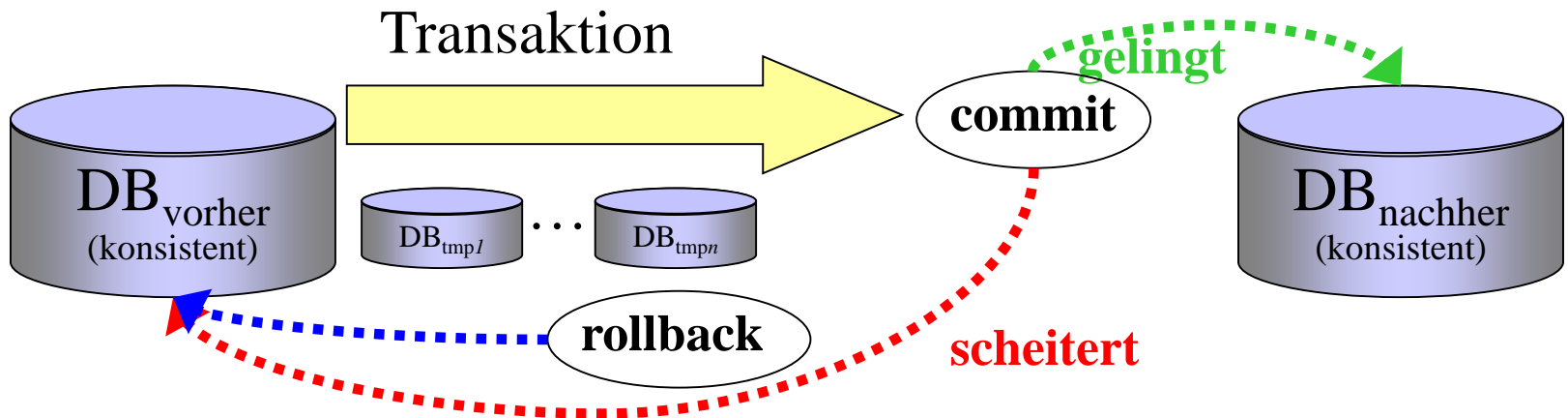
# Steuerung von Transaktionen

Unterstützung langer Transaktionen durch

- **define savepoint**
  - markiert einen zusätzlichen Sicherungspunkt, auf den sich die noch aktive Transaktion zurücksetzen lässt
  - Änderungen dürfen noch nicht festgeschrieben werden, da die Transaktion noch scheitern bzw. zurückgesetzt werden kann
  - SQL: `savepoint <identifier>`
- **backup transaction**
  - setzt die Datenbasis auf einen definierten Sicherungspunkt zurück
  - SQL: `rollback to <identifier>`

# Ende von Transaktionen

- **COMMIT gelingt**  
→ der neue Zustand wird dauerhaft gespeichert.
- **COMMIT scheitert**  
→ der ursprüngliche Zustand wie zu Beginn der Transaktion bleibt erhalten (bzw. wird wiederhergestellt). Ein COMMIT kann z.B. scheitern, wenn die Verletzung von Integritätsbedingungen erkannt wird.
- **ROLLBACK**  
→ Benutzer widerruft Änderungen



# Aufgaben eines DBMS

Wahrung eines korrekten DB-Zustands unter realen Benutzungsbedingungen, d.h.

1. **Synchronisation** (Concurrency Control)  
Schutz vor Fehlern durch sich gegenseitig störenden nebenläufigen Zugriff mehrerer Benutzer
2. **Datensicherheit** (Recovery)  
Schutz vor Verlust von Daten durch technische Fehler (Systemabsturz)
3. **Integrität** (Integrity)  
Schutz vor Verletzung der Korrektheit und Vollständigkeit von Daten durch *berechtigte* Benutzer



## Synchronisation (Concurrency Control)

- **Serielle Ausführung von Transaktionen**
  - 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

## Anomalien im Mehrbenutzerbetrieb

Wir unterscheiden u.a. folgende Grundmuster von Anomalien:

- Verloren gegangene Änderungen (Lost Updates)
  - Zugriff auf „schmutzige“ (nicht dauerhaft gültige) Daten (Dirty Read / 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

## 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`;
```

- Möglicher Ablauf

T1	T2
<pre>read(Passagiere.Gepäck, x1);  x1 := x1+3; write(Passagiere.Gepäck, x1);</pre>	<pre>read(Passagiere.Gepäck, x2); x2 := x2 + 5; write(Passagiere.Gepäck, x2);</pre>

- In der DB ist nur die Änderung von T1 wirksam, die Änderung von T2 ist verloren gegangen  
→ Verstoß gegen **Durability**

## 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

- Möglicher Ablauf:

T1	T2
<pre>UPDATE Passagiere SET Gepäck = Gepäck+3;  ROLLBACK;</pre>	<pre>UPDATE Passagiere SET Gepäck = Gepäck+5; COMMIT;</pre>

- 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

## 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

- Möglicher Ablauf:

T1	T2
<pre>SELECT Gepäck FROM Passagiere WHERE FlugNr = „BA932“;  SELECT Gepäck FROM Passagiere WHERE FlugNr = „BA932“;</pre>	<pre>INSERT INTO Passagiere VALUES (BA932, Meier, 3F, 5); COMMIT;</pre>

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

→ Verstoß gegen **Isolation**



## Phantomproblem

- Spezialfall des nicht-reproduzierbaren Lesens, bei der neu generierte Daten, sowie meist bei der 2. TA Aggregationsfunktionen 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

- Möglicher Ablauf

T1	T2
<pre>SELECT * FROM Passagiere WHERE FlugNr = „LH745“;  SELECT COUNT(*) FROM Passagiere WHERE FlugNr = „ LH745“;</pre>	<pre>INSERT INTO Passagiere VALUES (LH745, Phantomas, 7D, 2); COMMIT;</pre>

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

# 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

## Schedules

- Allgemeiner Schedule:  
Ein Schedule („Historie“) für eine Menge  $\{T_1, \dots, T_n\}$  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?

# Serialisierbarkeit von Transaktionen

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

***Kompromiss zwischen Performanz und Isolation (bzw. allem. und seriellen Schedules):***

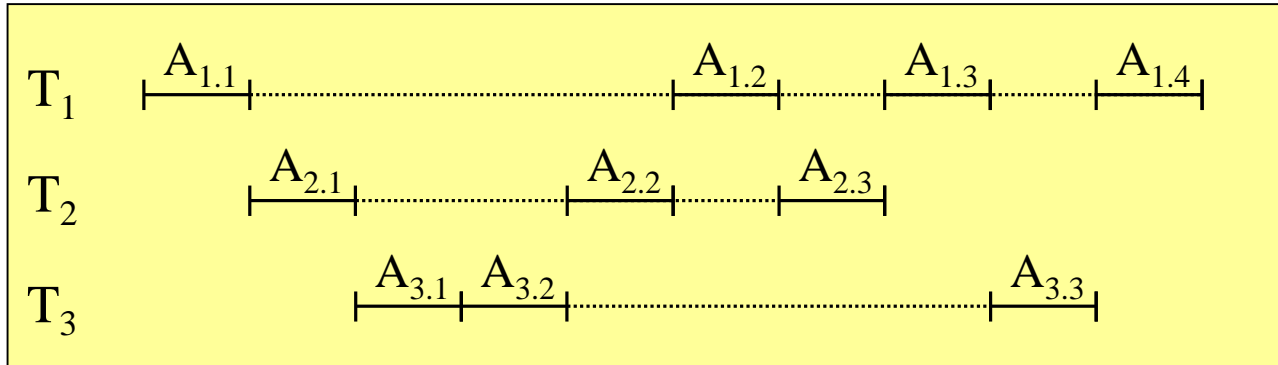
***Serialisierbarer Schedule:***

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

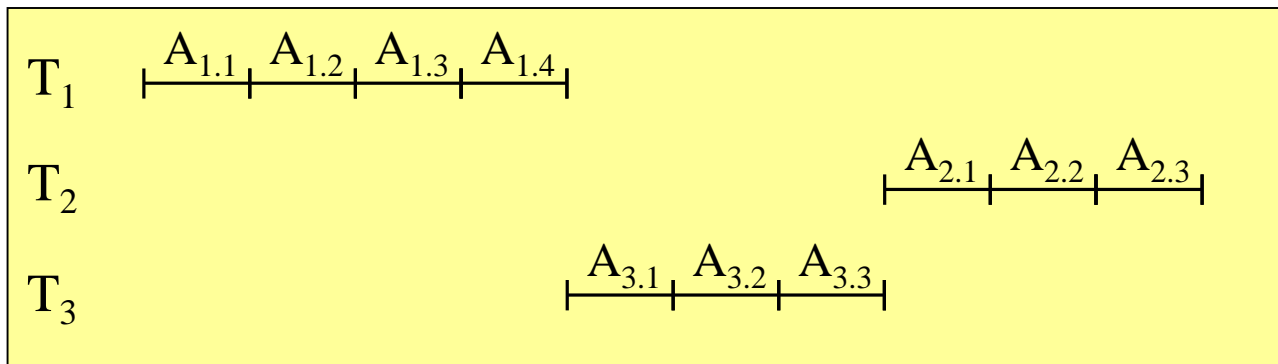
- Nur serialisierbare Schedules dürfen zugelassen werden!

# Serialisierbarkeit von Transaktionen

- Beispiele
  - Beliebiger Schedule:



- Serieller Schedule:



## Wirkung von Schedules

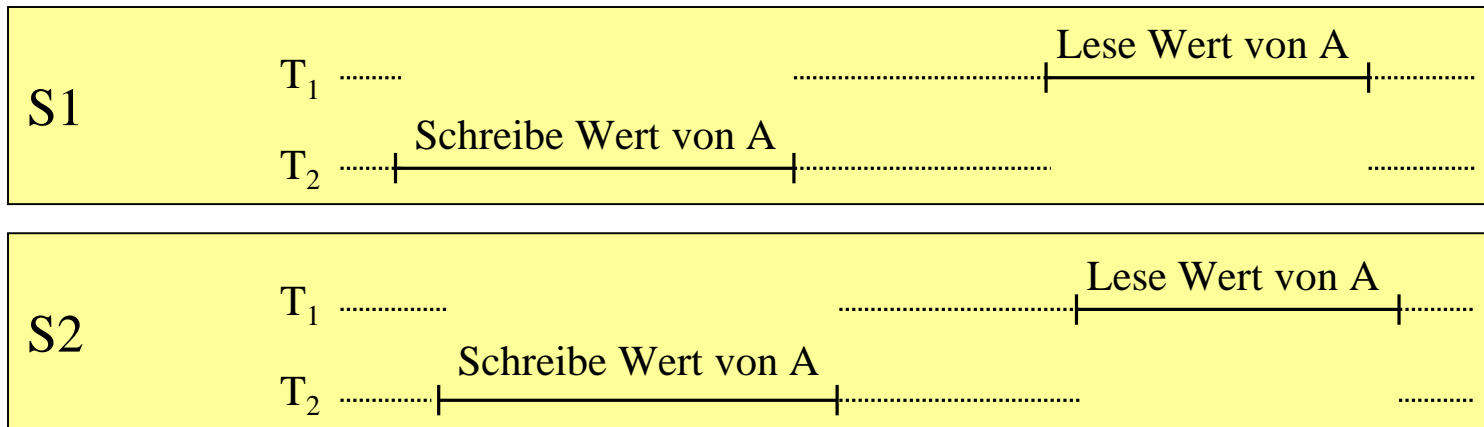
- Frage: Wann haben zwei Schedules S1 und S2 die gleiche Wirkung auf den Datenbank-Inhalt?
- Achtung:
  - Gleiches Ergebnis kann u.a. Ergebnis eines Zufalls sein
  - Dies könnte aber nur durch nachträgliches Überprüfen der Datenbank-Zustände nach S1 und S2 festgestellt werden.

# 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)



# Serialisierbarkeit von Transaktionen

## 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  $T_i \rightarrow T_j$ 
  - 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?)

- 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  $S_1$ ,  $S_2$  und  $S_3$  sind identisch
- Abhängigkeitsmengen:

$A_{S_1} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{2,1}(x)\}$

$A_{S_2} = \{rw_{2,1}(x), rw_{1,2}(x), ww_{2,1}(x)\}$

$A_{S_3} = \{rw_{1,2}(x), rw_{2,1}(x), ww_{1,2}(x)\}$

- Schedule  $S_1$  und  $S_2$  sind konfliktäquivalent
- Schedule  $S_1$  und  $S_3$ , bzw.  $S_2$  und  $S_3$  sind nicht konfliktäquivalent
- Schedule  $S_4$  ist kein Schedule derselben Transaktionen, da die Aktionen transaktionsintern vertauscht sind.

# Serialisierbarkeit von Transaktionen

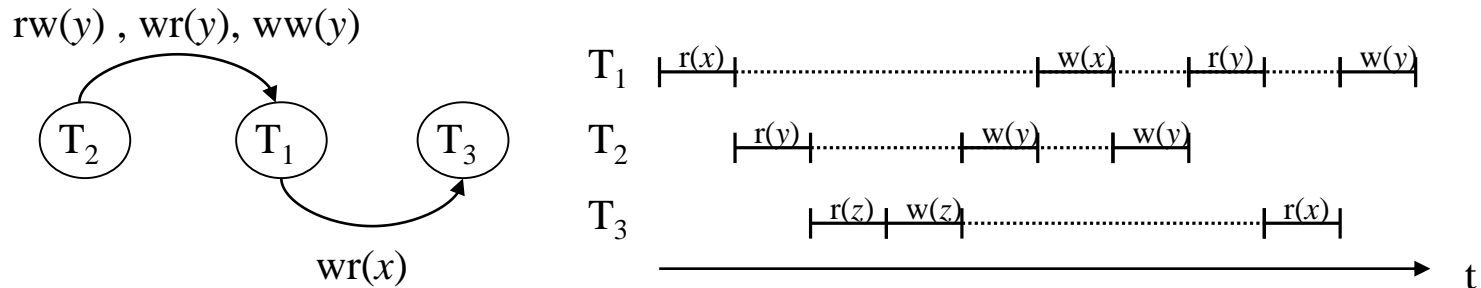
## Serialisierungs-Graph

- Überprüfung, ob ein Schedule von  $\{T_1, \dots, T_n\}$  serialisierbar ist (d.h. ob ein konflikt-äquivalenter serieller Schedule existiert)
- Die beteiligten Transaktionen  $\{T_1, \dots, T_n\}$  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.

# 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:  $S = (r_1(x), r_2(y), r_3(z), w_3(z), w_2(y), w_1(x), w_2(y), r_1(y), r_3(x), w_1(y))$

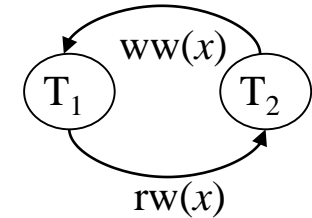
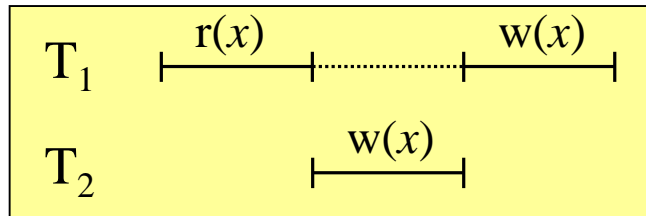


Serialisierungsreihenfolge:  $(T_2, T_1, T_3)$

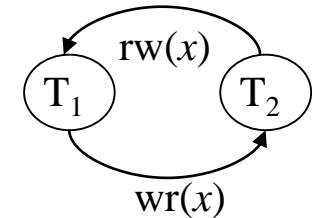
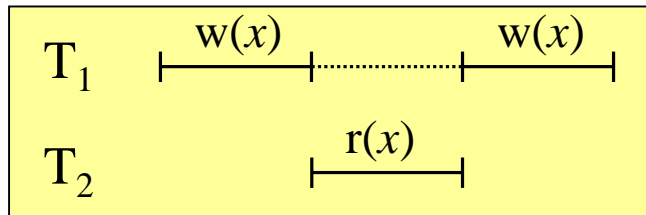
# Serialisierbarkeit von Transaktionen

## Beispiele für nicht-serialisierbare Schedules

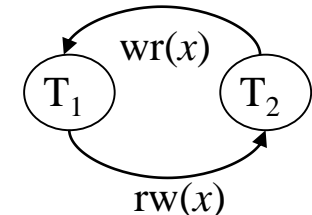
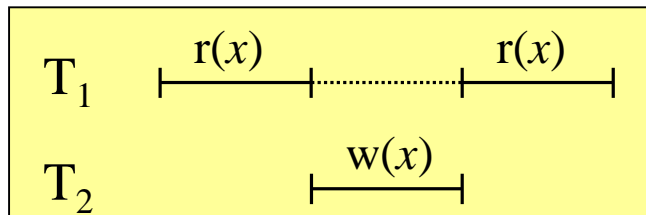
Lost Update:  $S=(r_1(x), w_2(x), w_1(x))$



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

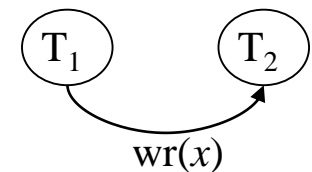
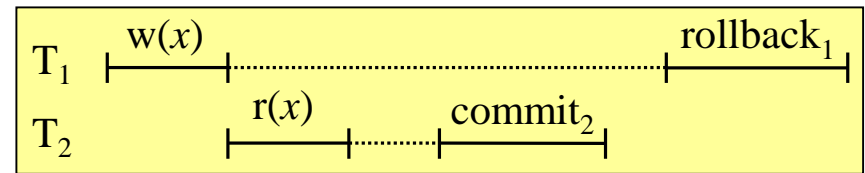


Non-repeatable Read:  $S=(r_1(x), w_2(x), r_1(x))$



## 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  $x$
  - $T_2$  führt *COMMIT* aus
  - Schedule ist serialisierbar, der Serialisierungs-Graph ist zyklensfrei
- 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



# 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



# 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 <sub>1</sub>				

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

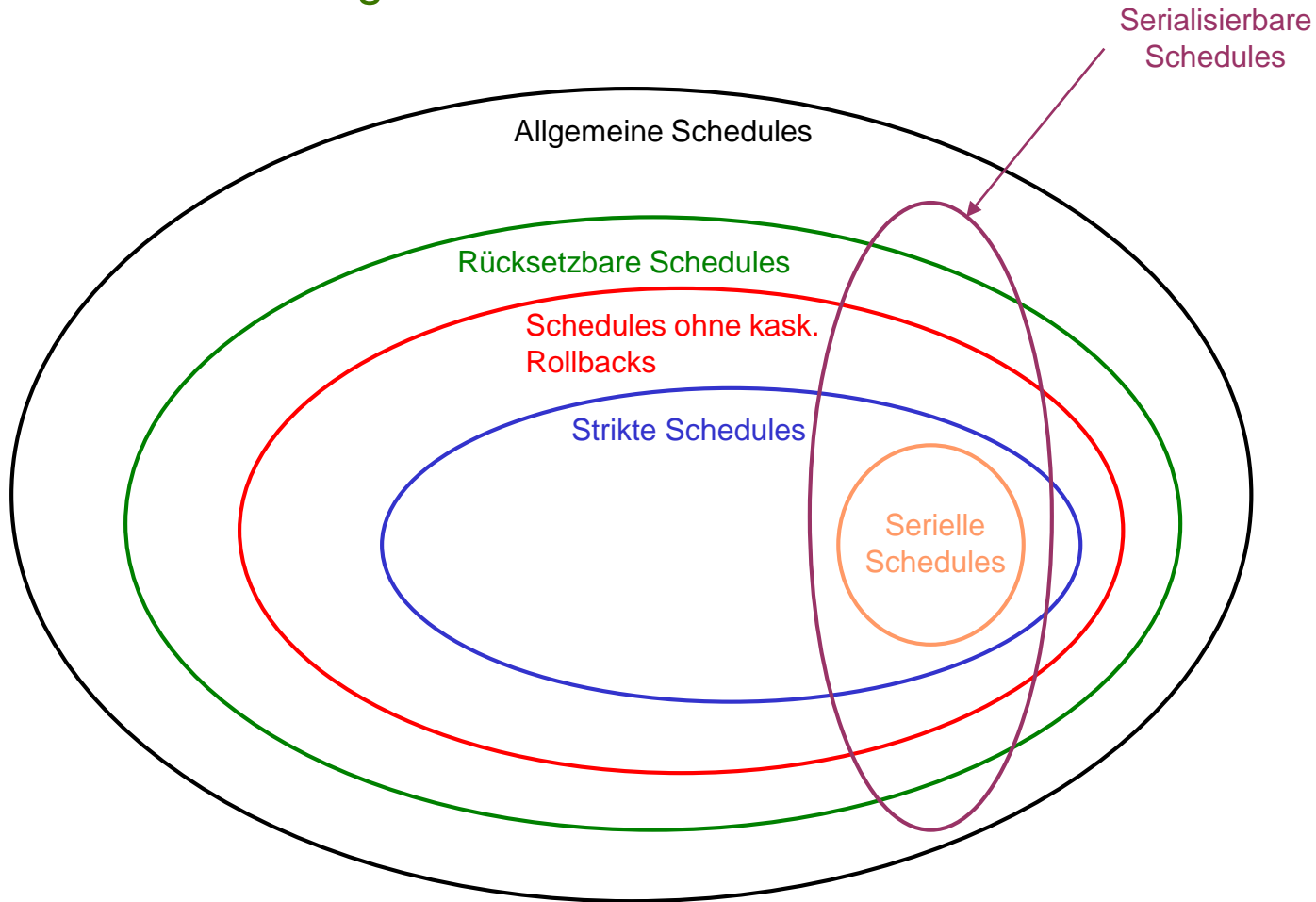
# Serialisierbarkeit von Transaktionen

## 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

# Serialisierbarkeit von Transaktionen

- Überblick: Beziehungen zwischen Scheduleklassen



# Serialisierbarkeit von Transaktionen

## Techniken zur Synchronisation

- Verwaltungsaufwand für Serialisierungsgraphen ist in der Praxis zu hoch. Deshalb: Andere Verfahren, die Serialisierbarkeit gewährleisten
- Pessimistische Ablaufsteuerung (Standardverfahren: Locking)
  - Konflikte werden vermieden, indem Transaktionen (typischerweise durch Sperren) blockiert werden
  - Nachteil: ggf. lange Wartezeiten
  - Vorteil: I.d.R. nur wenig Rücksetzungen aufgrund von Synchronisationsproblemen nötig
- Optimistische Ablaufsteuerung
  - Transaktionen werden im Konfliktfall zurückgesetzt
  - Transaktionen arbeiten bis zum COMMIT ungehindert. Anschließend erfolgt Prüfung (z.B. anhand von Zeitstempeln), ob Konflikt aufgetreten ist
  - Nur geeignet, falls Konflikte zwischen Schreibern eher selten auftreten

# Sperrverfahren (Locking)

## Allgemeines:

- Sperrverfahren sind DAS Standardverfahren zur Synchronisation in relationalen DBMS

## Sperre (Lock)

- Temporäres Zugriffsprivileg auf einzelnes DB-Objekt
- Anforderung einer Sperre durch *LOCK*, z.B. *L(x)* für *LOCK* auf Objekt *x*
- Freigabe durch *UNLOCK*, z.B. *U(x)* für *UNLOCK* von Objekt *x*
- *LOCK* / *UNLOCK* erfolgt atomar (also nicht unterbrechbar!)
- Sperrgranularität (Objekte, auf denen Sperren gesetzt werden): Datenbank, DB-Segment, Relation, Index, Seite, Tupel, Spalte, Attributwert
- Sperrenverwalter führt Tabelle für aktuell gewährte Sperren

# Sperrverfahren (Locking)

## Legale Schedules

- Vor jedem Zugriff auf ein Objekt wird eine geeignete Sperre gesetzt.
- Keine Transaktion fordert eine Sperre an, die sie schon besitzt.
- Spätestens bei Transaktionsende werden alle Sperren zurückgegeben.
- Sperren werden respektiert, d.h. eine mit gesetzten Sperren unverträgliche Sperranforderung (z.B. exklusiver Zugriff auf Objekt x) muss warten.

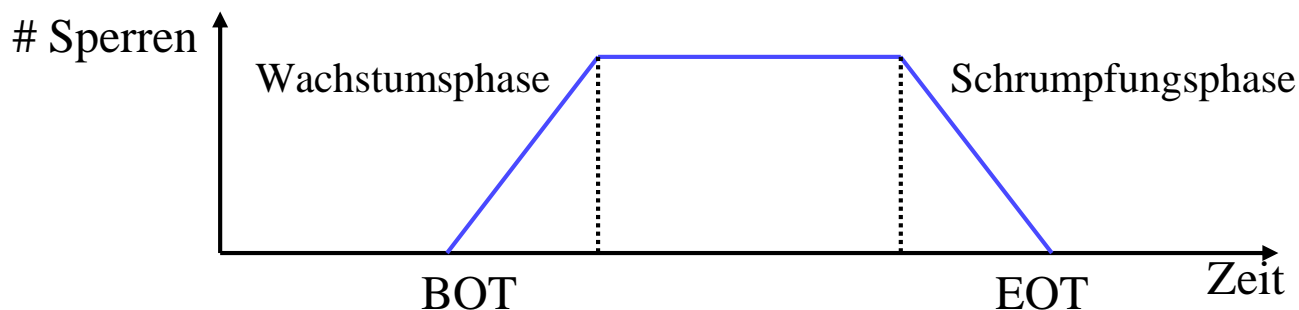
## Bemerkungen

- Anfordern und Freigeben von Sperren sollte das DBMS implizit selbst vornehmen.
- Die Verwendung legaler Schedules garantiert noch nicht die Serialisierbarkeit.

# Sperrverfahren (Locking)

## Zwei-Phasen-Sperrprotokoll (2PL)

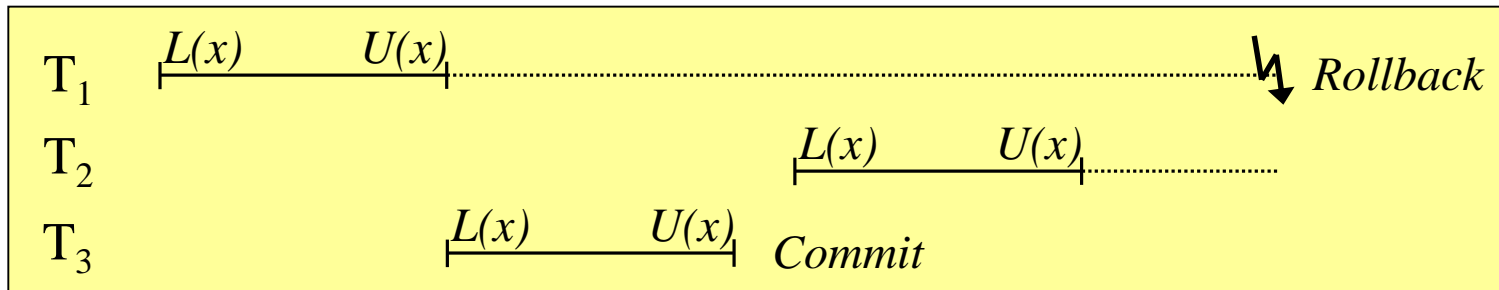
- Einfachste und gebräuchlichste Methode, um ausschließlich serialisierbare Schedules zu erzeugen
- Merkmal: keine Sperrenfreigabe vor der letzten Sperrenanforderung einer Transaktion
- Ergebnis: Ablauf in zwei Phasen
  - Wachstumsphase: Anforderungen der Sperren
  - Schrumpfungsphase: Freigabe der Sperren



# Sperrverfahren (Locking)

## Zwei-Phasen-Sperrprotokoll (2PL)

- Serialisierbarkeit ist gewährleistet, da Serialisierungsgraphen keine Zyklen enthalten können ☺
- Problem : Gefahr des kaskadierenden Rücksetzens im Fehlerfall (bzw. sogar **nicht-rücksetzbar**) ☹

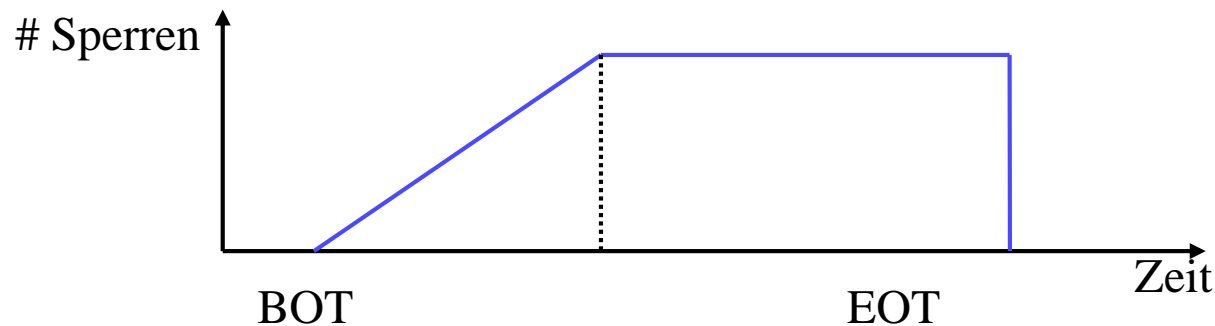


- Transaktion  $T_1$  wird nach  $U(x)$  zurückgesetzt
- $T_2$  hat “schmutzig” gelesen und muss zurückgesetzt werden
- Sogar  $T_3$  muss zurückgesetzt werden  
→ Verstoß gegen die Dauerhaftigkeit (ACID) des **COMMIT**!



## Striktes Zwei-Phasen-Sperrprotokoll

- Abhilfe durch striktes (oder strenges) Zwei-Phasen-Sperrprotokoll:
  - Alle Sperren werden bis zum *COMMIT* gehalten
  - *COMMIT* wird atomar (d.h. nicht unterbrechbar) ausgeführt



# Sperrverfahren (Locking)

## Erhöhung des Parallelisierungsgrads

- Striktes 2PL erzwingt serialisierbare, rücksetzbare Schedules
- ABER: Parallelität der TAs wird dadurch stark eingeschränkt
  - Objekt ist entweder gesperrt (und dann bis zum Commit der entspr. TA) oder zur Bearbeitung frei
    - => kein paralleles Lesen oder Schreiben möglich
- Beobachtung: Parallelität unter Lesern könnte man eigentlich erlauben, da hier die Isoliertheit der beteiligten TAs nicht verletzt wird
- Daher statt 1 nun 2 Arten von Sperren
  - Lesesperren oder R-Sperren (read locks)
  - Schreibsperren oder X-Sperren (exclusive locks)

## RX-Sperrverfahren

- R- und X-Sperren
- Parallelität unter Lesern erlaubt
- Verträglichkeit der Sperrentypen (siehe Tabelle rechts)

		<i>bestehende Sperre</i>	
		<b>R</b>	<b>X</b>
<i>angeforderte Sperre</i>	<b>R</b>	+	-
	<b>X</b>	-	-

## Serialisierungsreihenfolge bei RX

- RX-Sperrverfahren meist in Verbindung mit striktem 2PL um nur kaskadenfreie rücksetzbare Schedules zu erhalten
- Zur Erinnerung: Die Reihenfolge der Transaktionen im „äquivalenten seriellen Schedule“ ist die Serialisierungsreihenfolge.
- Bei RX-Sperrverfahren (in Verbindung mit striktem 2PL) wird die Serialisierungsreihenfolge durch die erste auftretende Konfliktoperation festgelegt.

# Sperrverfahren (Locking)

- Beispiel (Serialisierungsreihenfolge bei RX):
  - Situation:
    - $T_1$  schreibt ein Objekt  $x$
    - Danach möchte  $T_2$  Objekt  $x$  lesen
  - Folge:
    - $T_2$  muss auf das *COMMIT* von  $T_1$  warten, d.h. der serielle Schedule enthält  $T_1$  vor  $T_2$ .
    - Da  $T_2$  wartet, kommen auch alle weiteren Operationen erst nach dem *COMMIT* von  $T_1$ .
  - Achtung:

Grundsätzlich sind zwar auch Abhängigkeiten von  $T_2$  nach  $T_1$  denkbar (z.B. auf einem Objekt  $y$ ), diese würden aber zu einer **Verklemmung** (**Deadlock**, gegenseitiges Warten) führen.