

# Datenbanken I (T2INF2004)

## Foliensatz 8: Mehrbenutzersysteme (ACID- Prinzip, Recovery, Transaktionen)

Uli Seelbach, DHBW Mannheim, 2023

Vereinzelte Bildquellen in diesem Foliensatz von Heiko Weber,  
Hochschule Darmstadt, <http://weberhda.iwebadmin.de/> und Prof. Böhm  
<http://www.dbs.ifi.lmu.de/Lehre/DBSII/SS2007/Skript.2007/dbs2-02.pdf> bzw. 2009 und Prof. Manthey



# Transaktionen

## Definition

Eine oder mehrere Datenbankoperationen, die zu einer logischen Einheit zusammengefasst werden

- Ausführung ganz oder gar nicht
- Bei Fehler muss komplette Transaktion abgebrochen oder wiederholt werden
- D.h. nach abgebrochener Transaktion muss Zustand dem Ausgangszustand entsprechen
- Eine Transaktion kann auch nur aus einer einzigen Operation bestehen
- Jede einzelne (Änderungs-)Operation ist atomar (wird also ganz oder gar nicht ausgeführt).
- Im Rahmen des Studiums werden für Sie nur Transaktionen von DML-Abfragen relevant sein (es gibt auch DDL-Transaktionen)



# Transaktionen

## Operatoren

- Begin Of Transaction (BOT)
  - Anfang einer Transaktion
- Commit
  - Beendigung der Transaktion. Alle Änderungen der Datenbasis werden festgeschrieben, d.h. dauerhaft in die Datenbank eingebracht.
- Rollback (Abort)
  - Beendigung der Transaktion durch Selbstabbruch. Das DBMS macht alle Änderungen dieser Transaktion rückgängig.
- (Savepoints)
  - Sicherungspunkte innerhalb einer Transaktion



# Transaktionen

## ACID-Prinzip

- **A**tomicity
  - „Alles oder nichts“
- **C**onsistency
  - Datenbank muss sich nach Transaktionsende in einem konsistenten Zustand befinden
  - Referenzielle Integrität kann in einer Tx ggf. sogar verletzt sein
- **I**solation
  - Transaktionen dürfen sich gegenseitig nicht beeinflussen
  - In der Praxis gibt es mehrere Freiheitsgrade
- **D**urability
  - Die Aktionen einer erfolgreich abgeschlossenen Transaktionen dürfen auch bei Ausfall nicht verloren gehen



# Transaktionen

## Transaktionssteuerung in SQL - Grundlagen

- **commit [work]**

Die in der Transaktion vollzogenen Änderungen werden – falls keine Konsistenzverletzung oder andere Probleme aufgedeckt werden – festgeschrieben.

- **rollback [work]**

Alle Änderungen der Transaktion werden zurückgesetzt. Anders bei commit muss das DBMS die „erfolgreiche“ Ausführung eines rollback-Befehls immer garantieren können.

- Sowohl commit als auch rollback tun noch ein wenig mehr: Es werden Ressourcen der Tx (Speicherplatz, Sperren, ...) freigegeben
- Auch rein lesende Tx sollten daher committed (oder zurückgesetzt) werden



# Transaktionen

## Transaktionssteuerung in SQL - Grundlagen

- BOT

- Unterschied zwischen impliziten und expliziten Transaktionen
- je nach DBMS unterschiedlich
  - in Oracle implizit bei einem SQL-Stmt, und endet erst mit COMMIT; oder ROLLBACK;
  - DDL-Statements werden automatisch committet
- In MySQL und SQL Server gibt es standardmäßig Autocommits, somit muss beim Abweichen vom Standard eine T mit „START TRANSACTION“ / „BEGIN TRANSACTION“ eingeleitet werden

### Beispiel

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

# Fehlerbehandlung und Backup

## Fehlerarten in Transaktionen



?

**Welche Arten von Fehlern  
können im Betrieb einer DB  
auftreten?**



# Fehlerbehandlung und Backup

## Fehlerarten in Transaktionen

- **Transaktionsfehler**

- Lokaler Fehler in einer noch nicht festgeschriebenen (committed) Transaktion, Wirkung muss zurückgesetzt werden
- Unterschiedlichste Ursachen (FK passt nicht, Zahl zu groß, Programmabsturz, ...)
- R1-Recovery

- **Systemfehler** / Fehler mit Hauptspeicherverlust

- Abgeschlossene TAs müssen erhalten bleiben (R2-Recovery)
- Nicht abgeschlossene TAs müssen zurückgesetzt werden (R3-Recovery)

- **Externspeicherfehler**

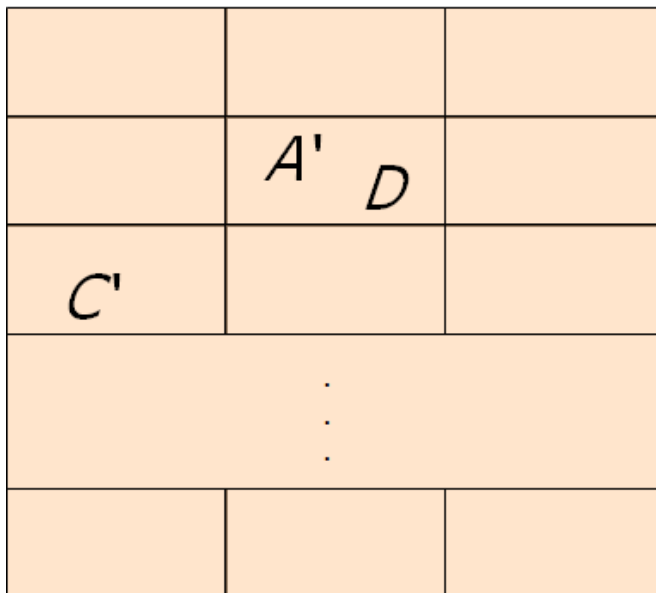
- Festplatte defekt, Katastrophenfall, menschliches Versagen
- R4-Recovery



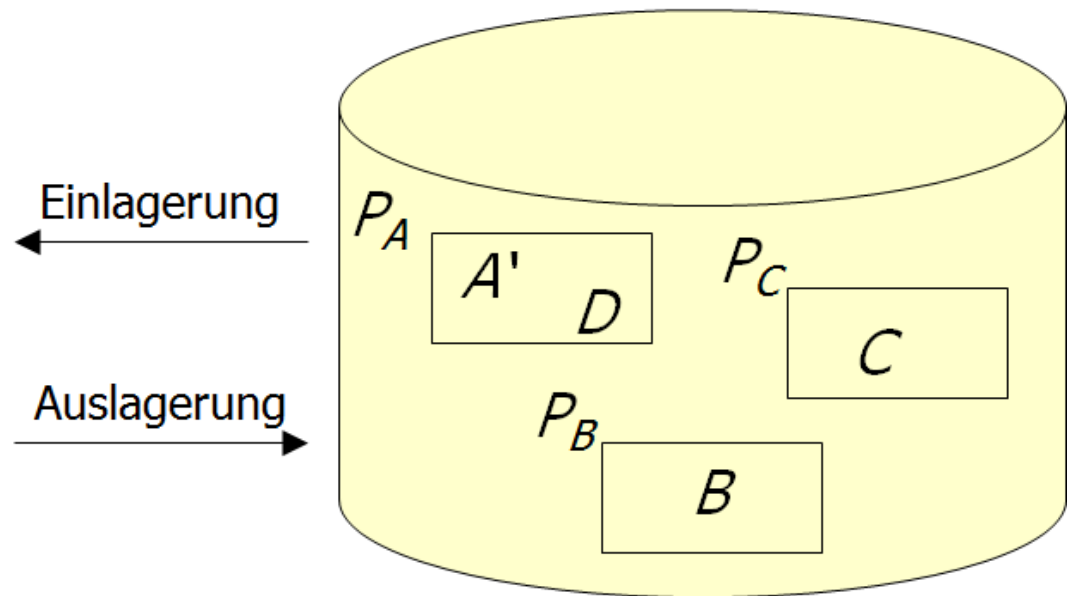
# Fehlerbehandlung und Backup

## Grundlegendes Problem: mehrstufige Speicherhierarchie

DBMS-Puffer

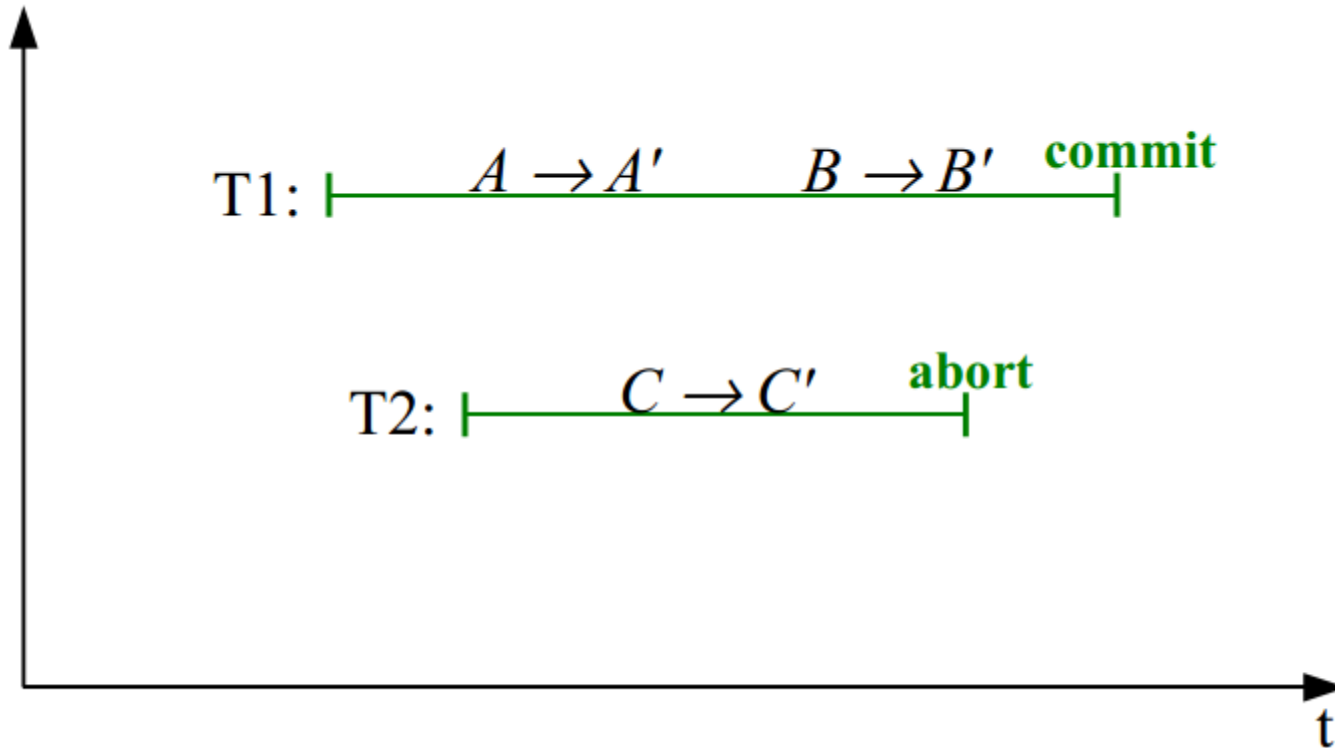


Hintergrundspeicher



# Transaktionsfehler (R1-Recovery)

## Szenario



- $C'$  muss auf  $C$  zurückgesetzt werden  $\rightarrow$  alle Datenänderungen in T2 werden auf den Zustand zum Start von T2 zurückgesetzt
  - Bei T1 werden Änderungen festgeschrieben



# Protokollierung von Datenänderungen

## Aufbau

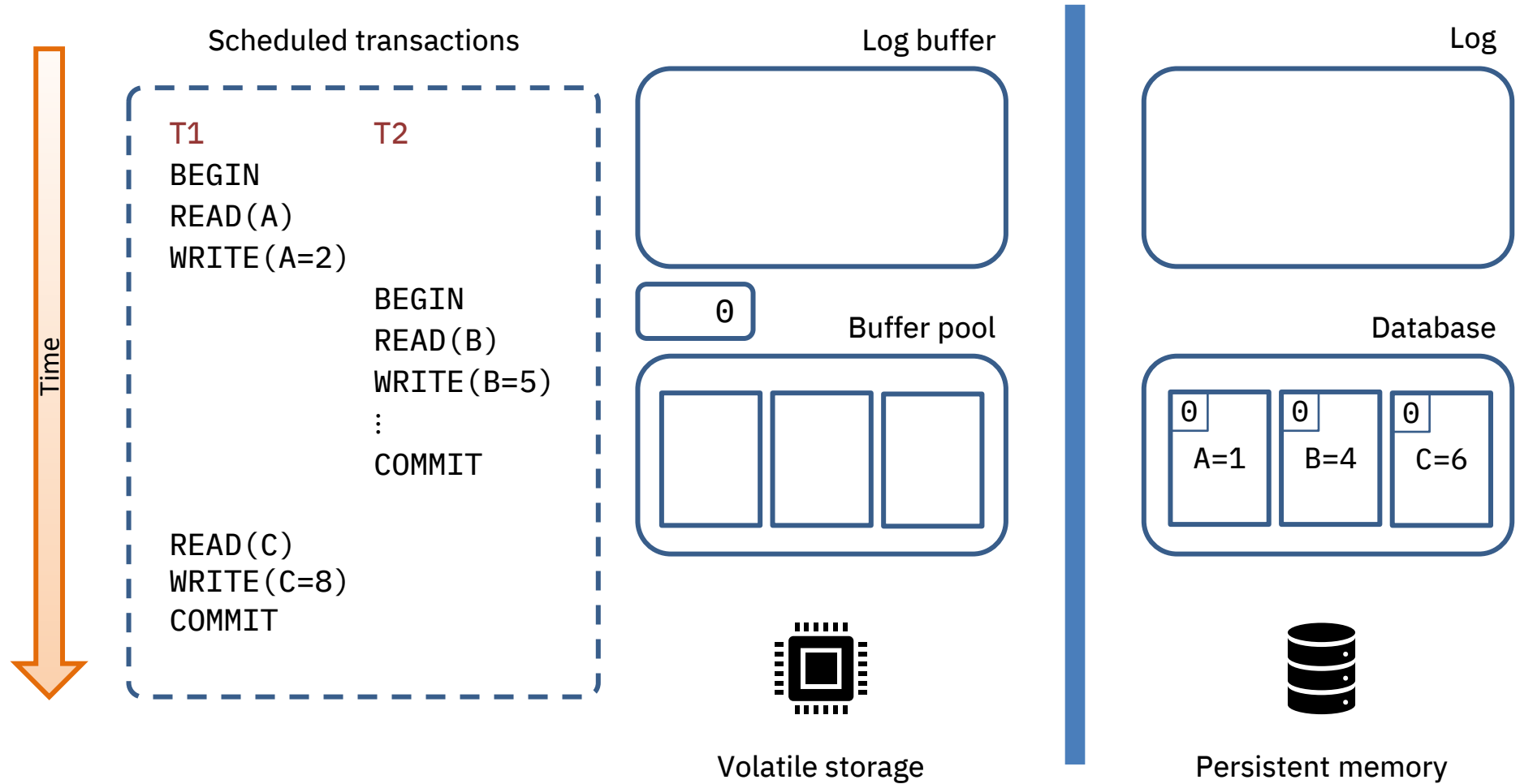
- [LSN, TA, PageID, Redo, Undo, PrevLSN]
- LSN (Log Sequence Number),
  - eine eindeutige Kennung des Log-Eintrags, monoton aufsteigend
  - chronologische Reihenfolge der Protokolleinträge kann so ermittelt werden
- TA
  - Transaktionskennung: Transaktion, die die Änderung durchgeführt hat
- PageID
  - die Kennung der Seite, auf der Änderungsoperation vollzogen wurde
  - Wenn eine Änderung mehr als eine Seite betrifft, müssen entsprechend viele Log-Einträge generiert werden

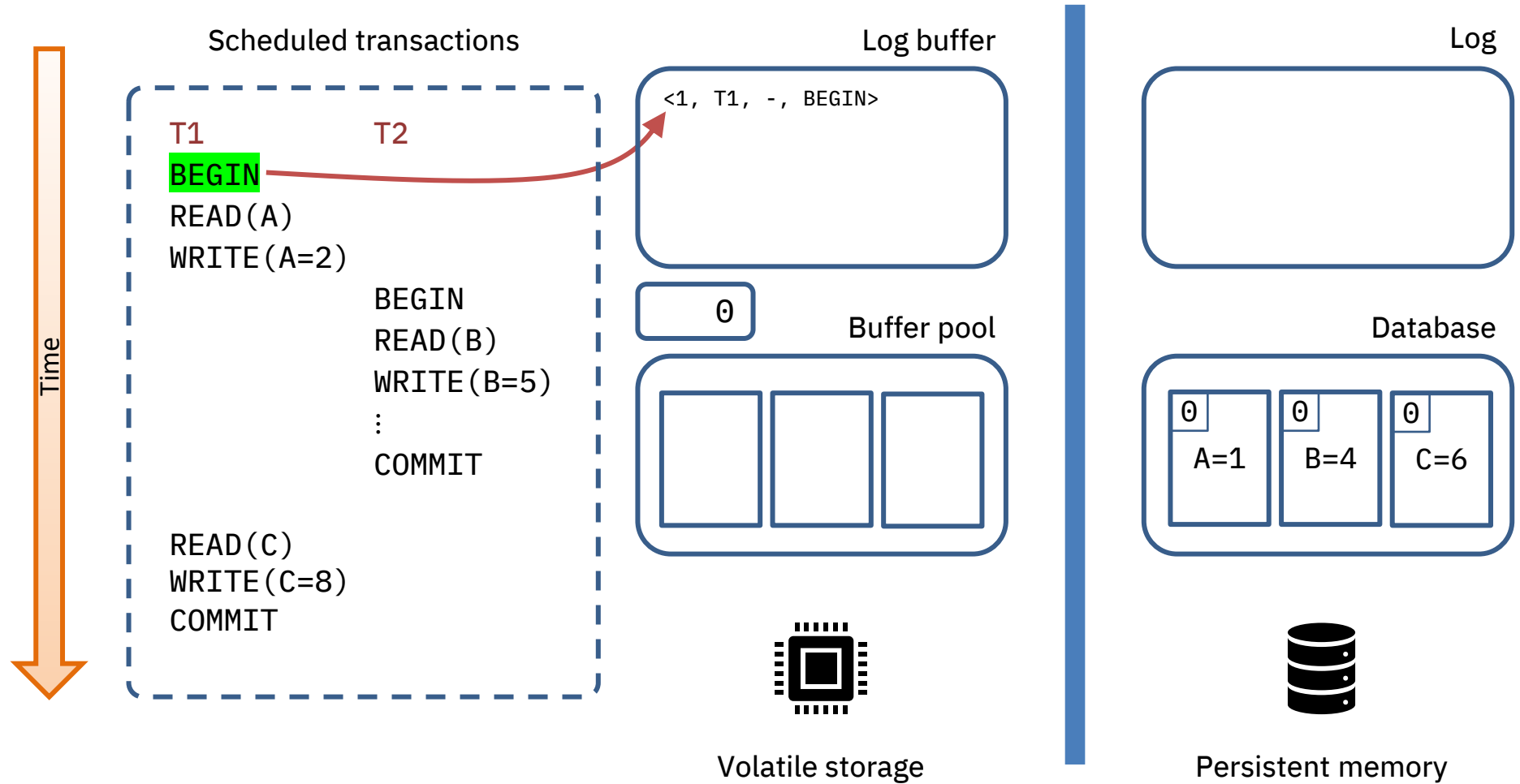


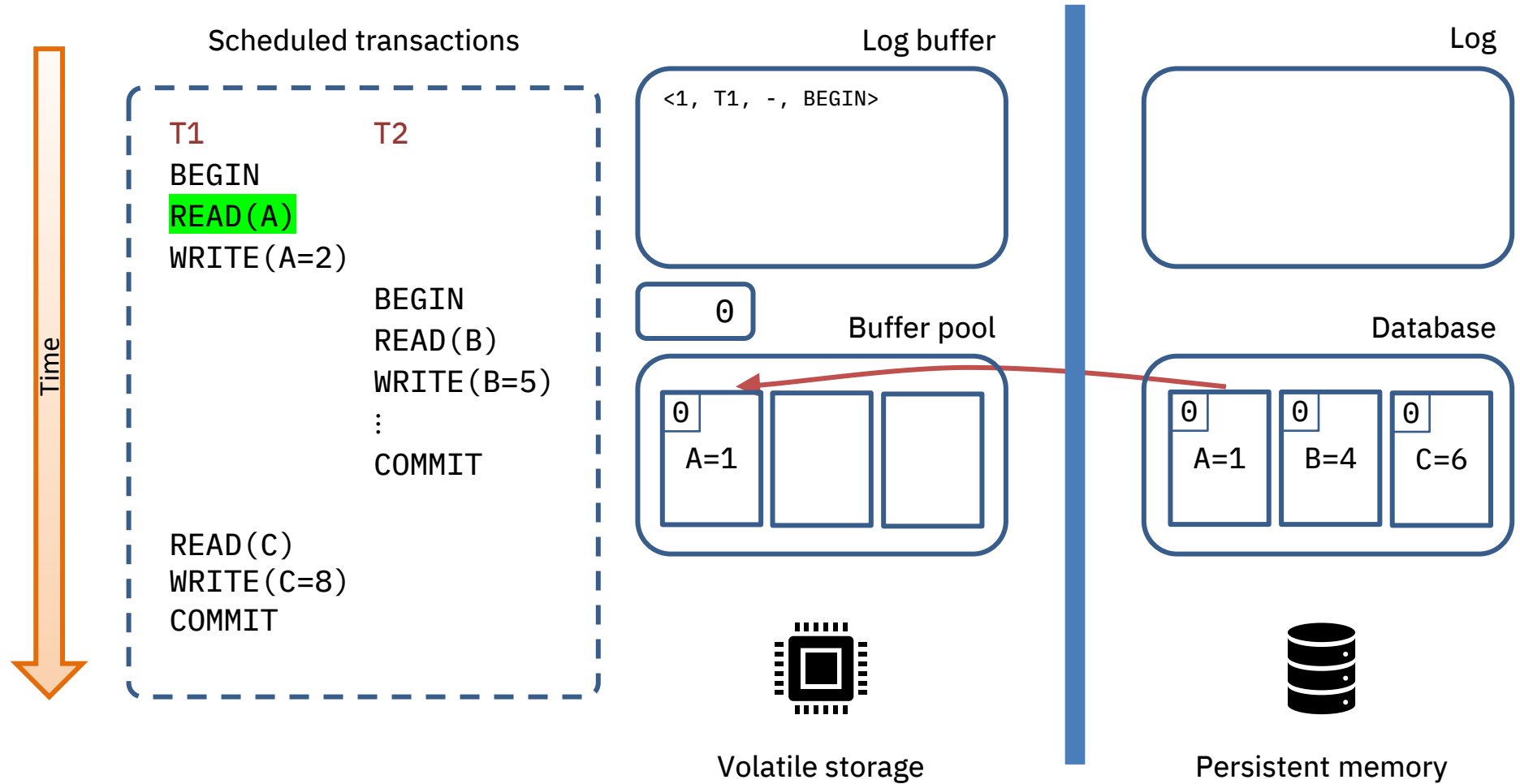
# Protokollierung von Datenänderungen

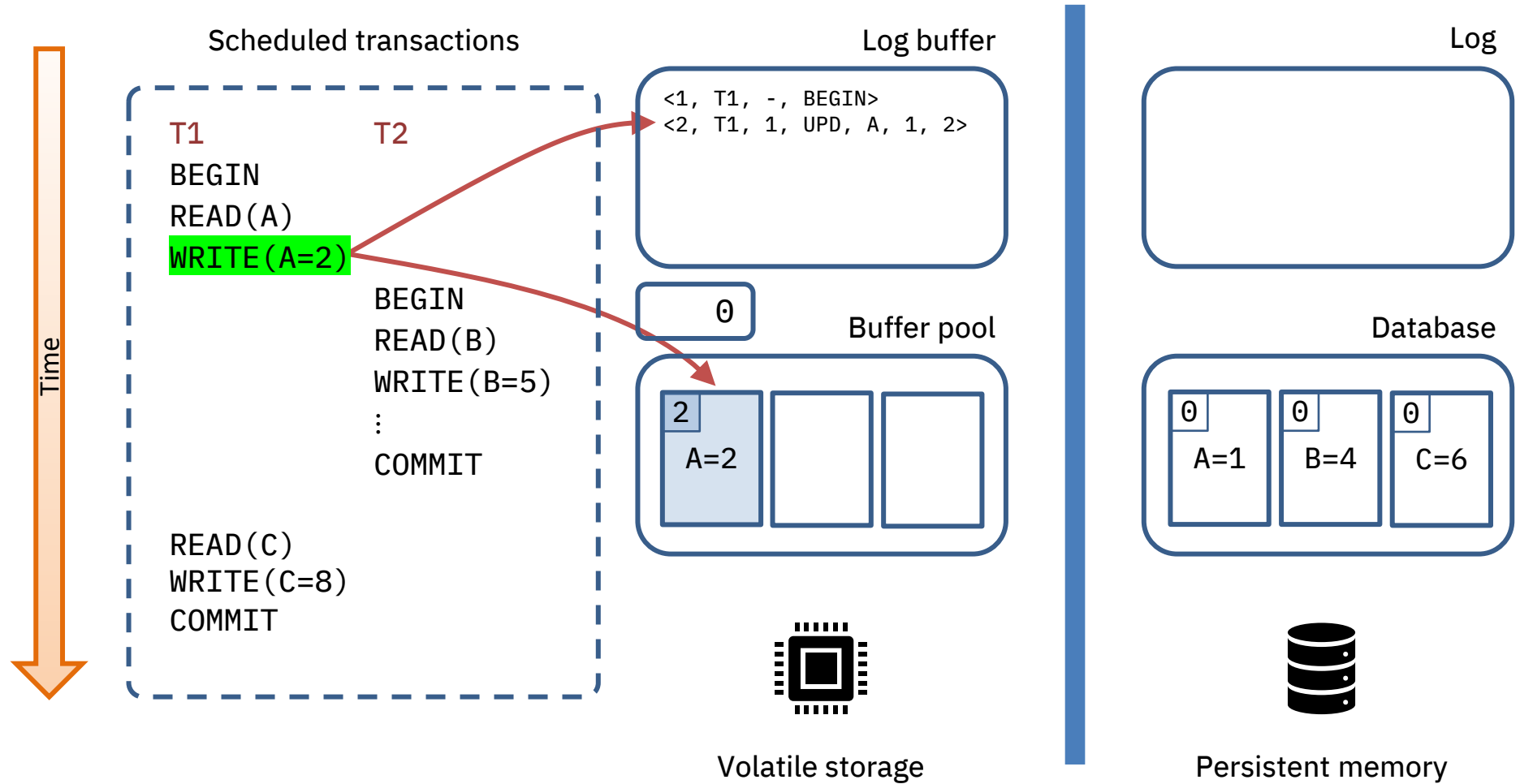
## Aufbau

- [ LSN, TA, PrevLSN, PageID, Redo, Undo ]
  - Die Redo-Information gibt an, wie die Änderung wiederholt werden kann
  - Die Undo-Information beschreibt, wie die Änderung rückgängig gemacht werden kann
  - PrevLSN, einen Zeiger auf den vorhergehenden Log-Eintrag der jeweiligen Transaktion. Redundant, aber aus Effizienzgründen sinnvoll
  - Zusätzlich werden BOT, commit und abort auch protokolliert
    - PageID, Redo und Undo werden hier nicht benötigt
- Zu jedem „DO“ gibt es also ein „UNDO“ und ein „REDO“

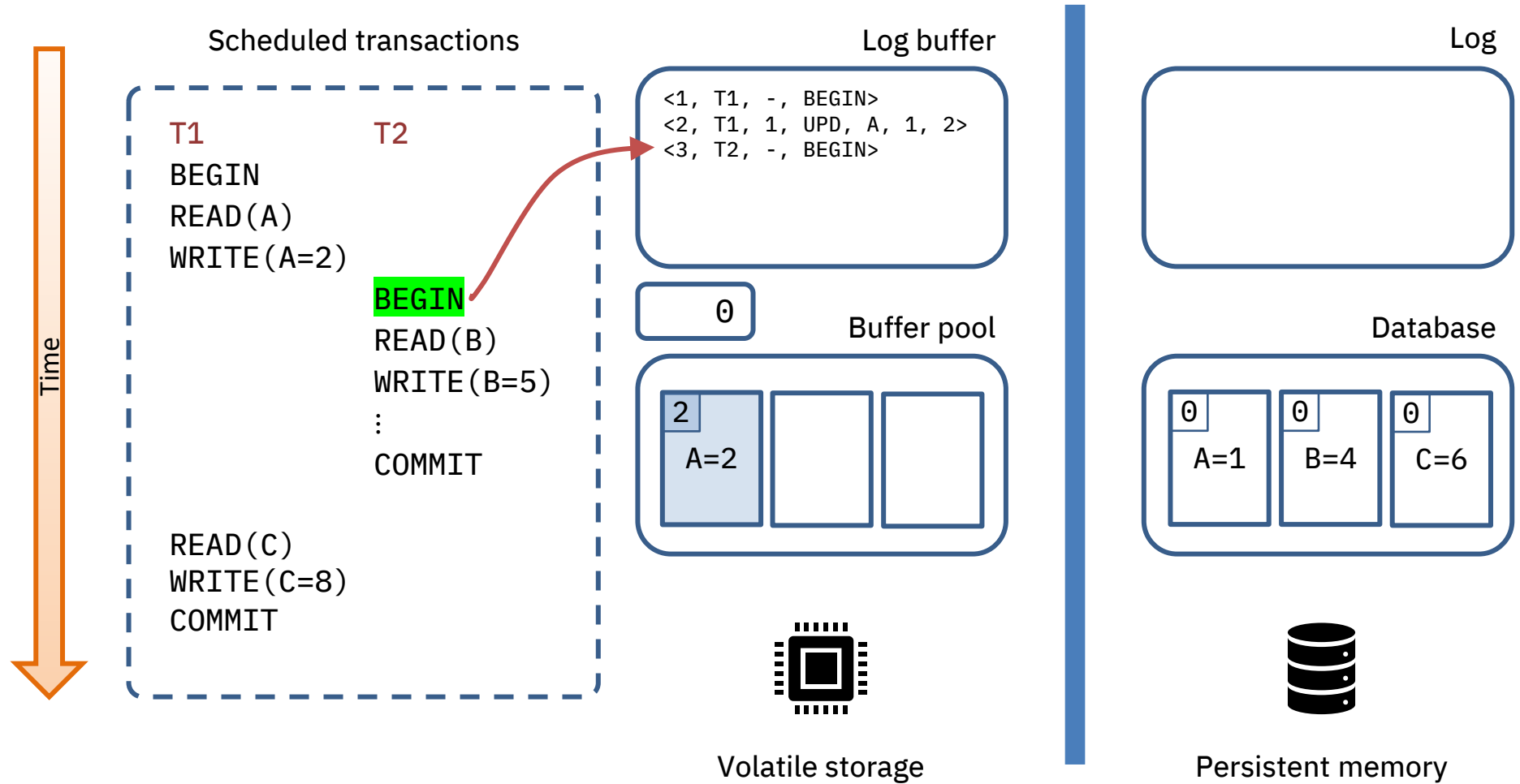


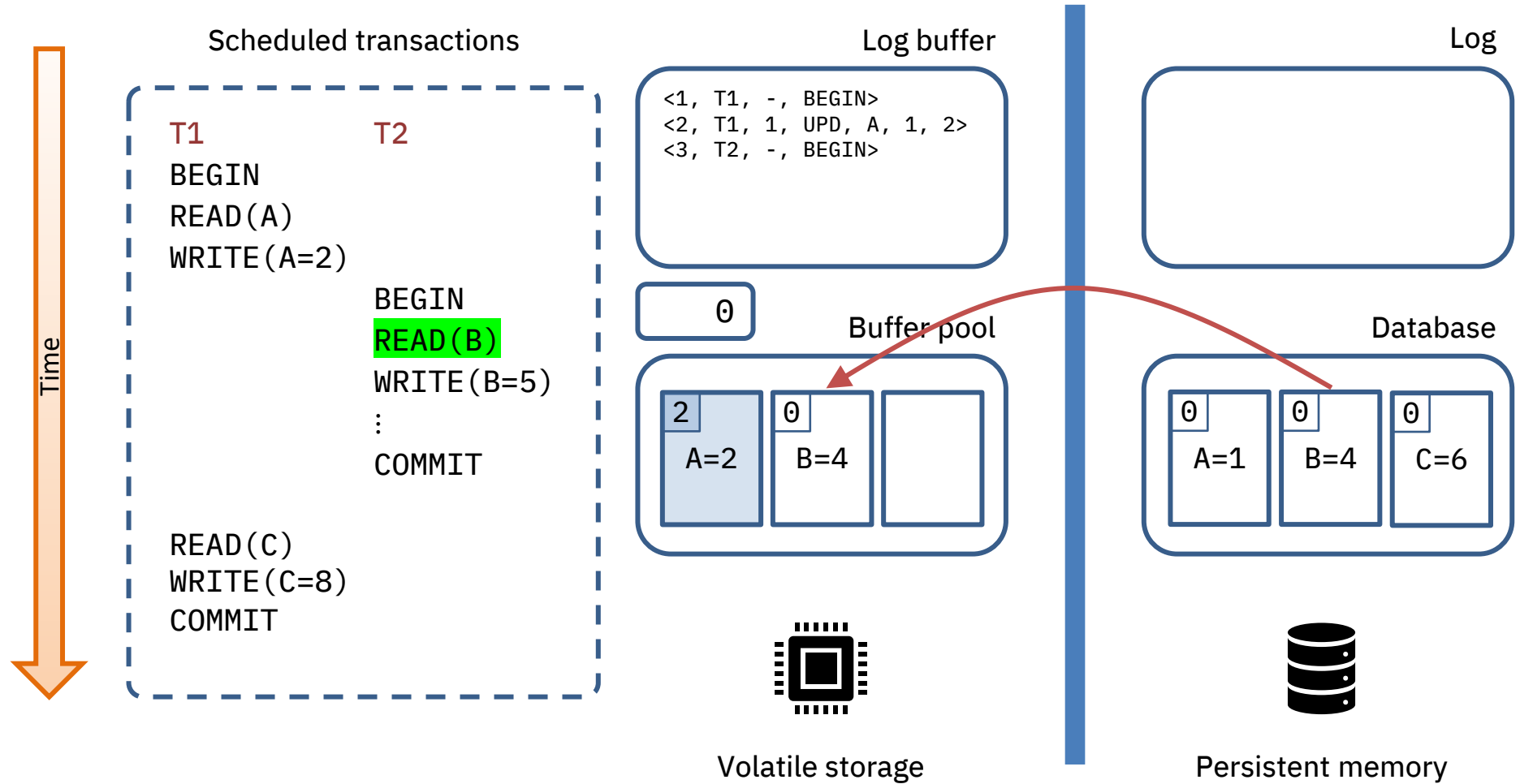


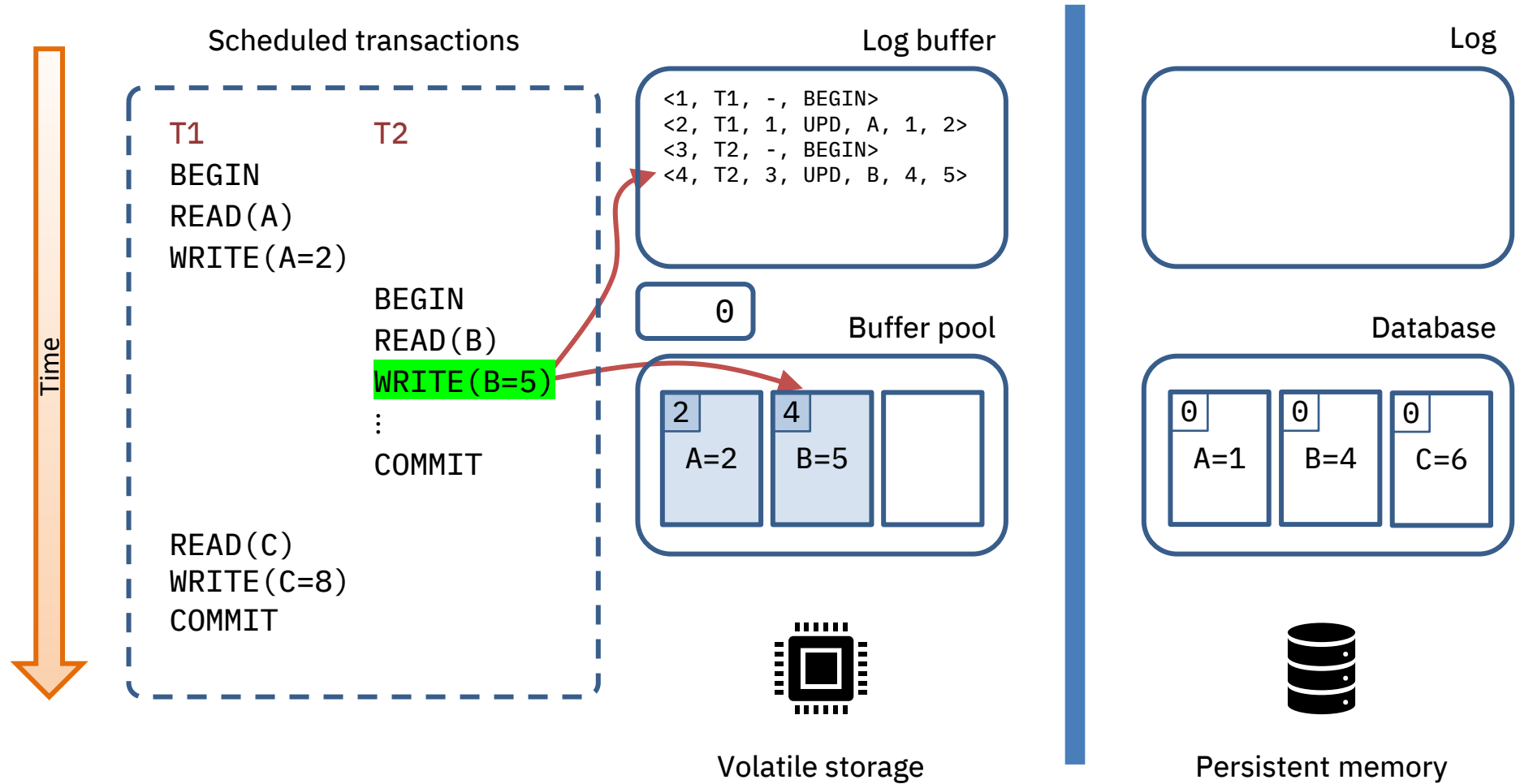


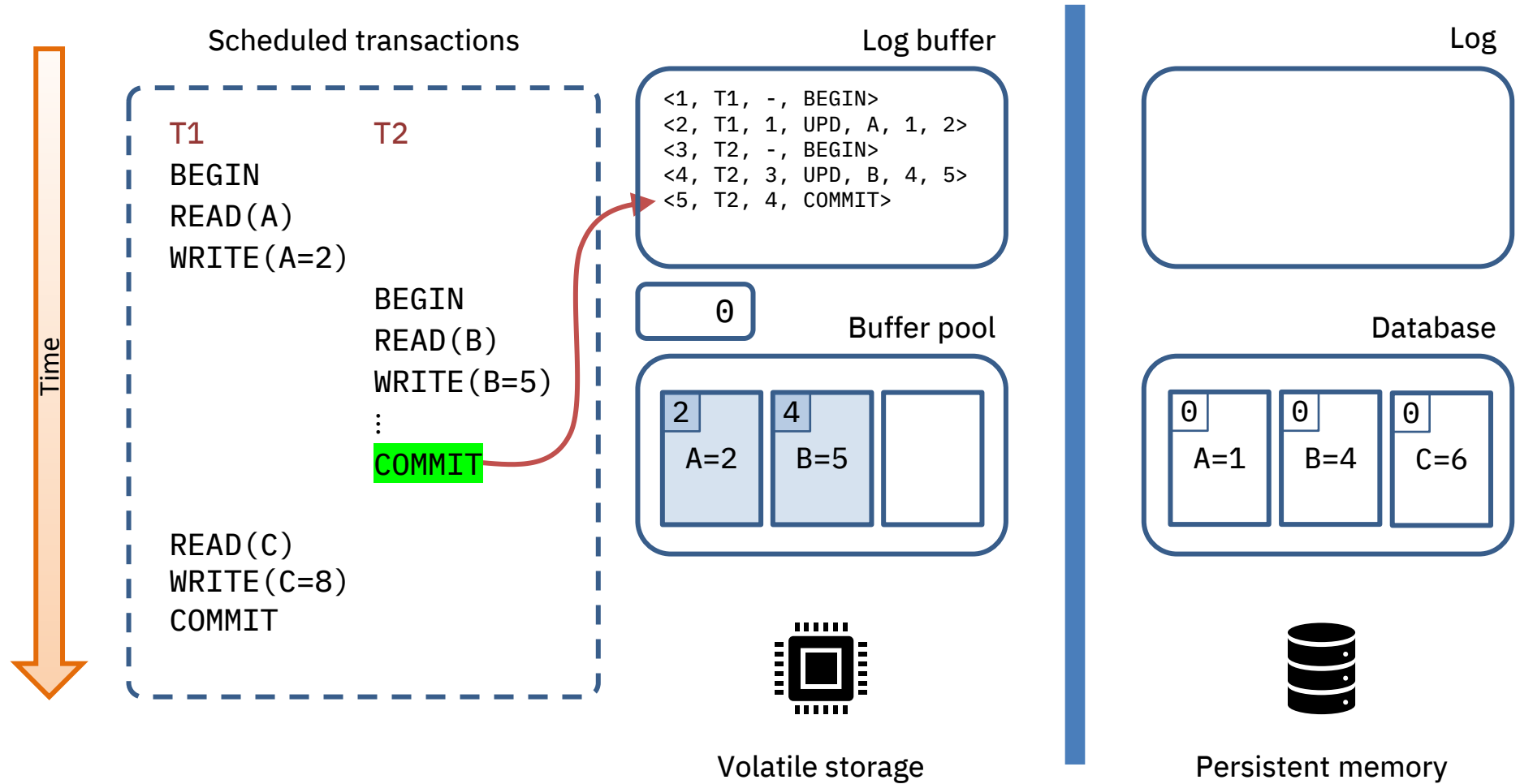


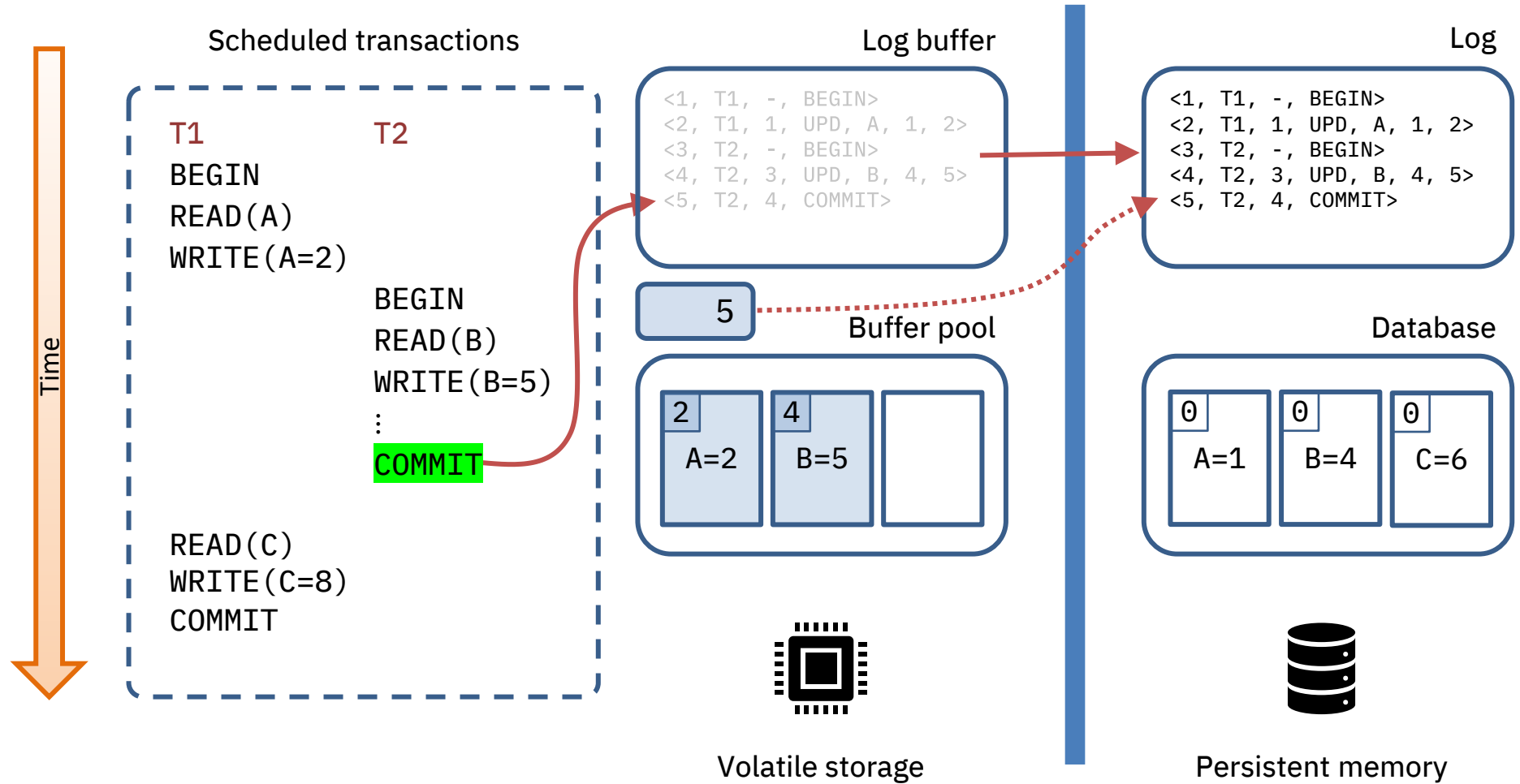


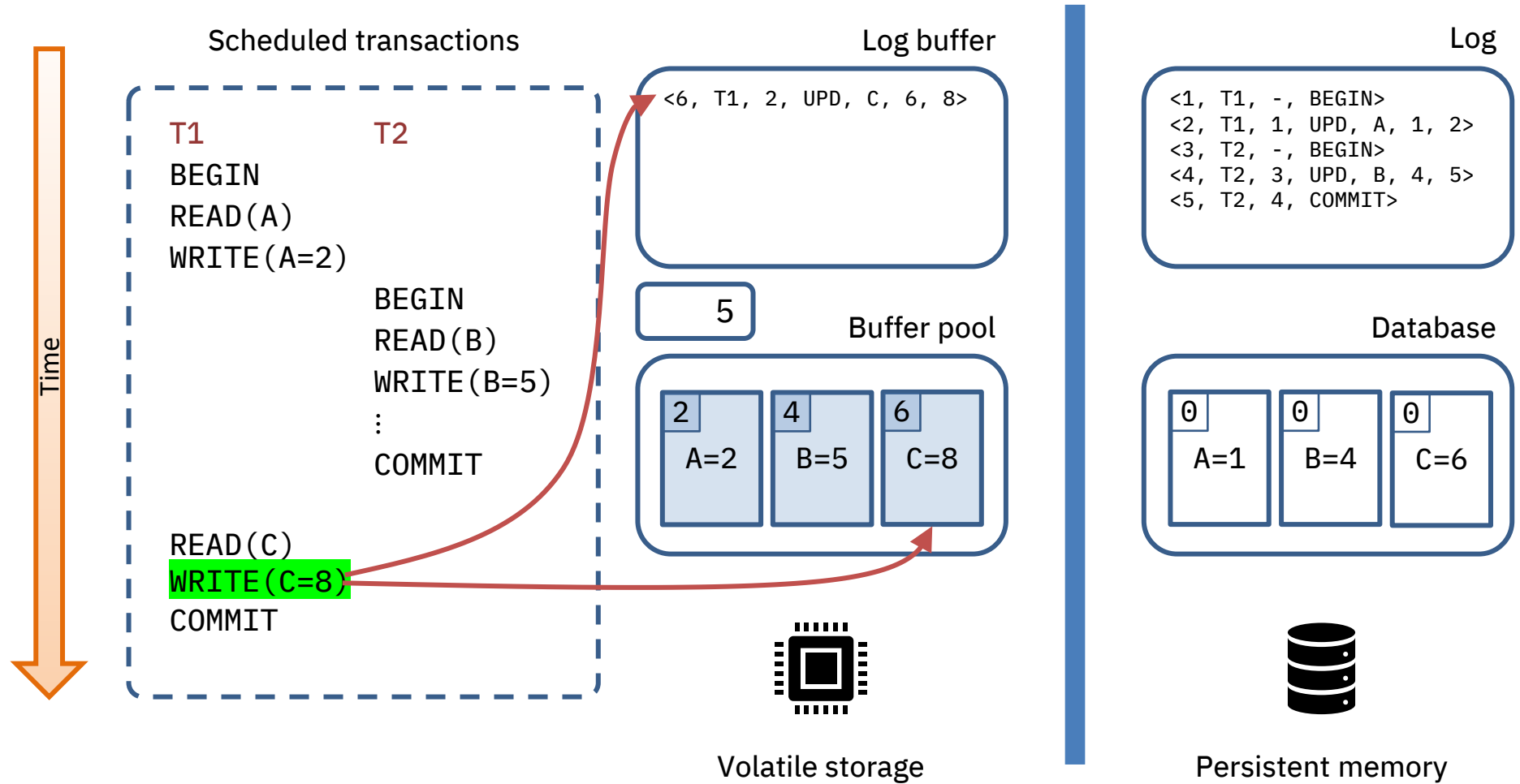


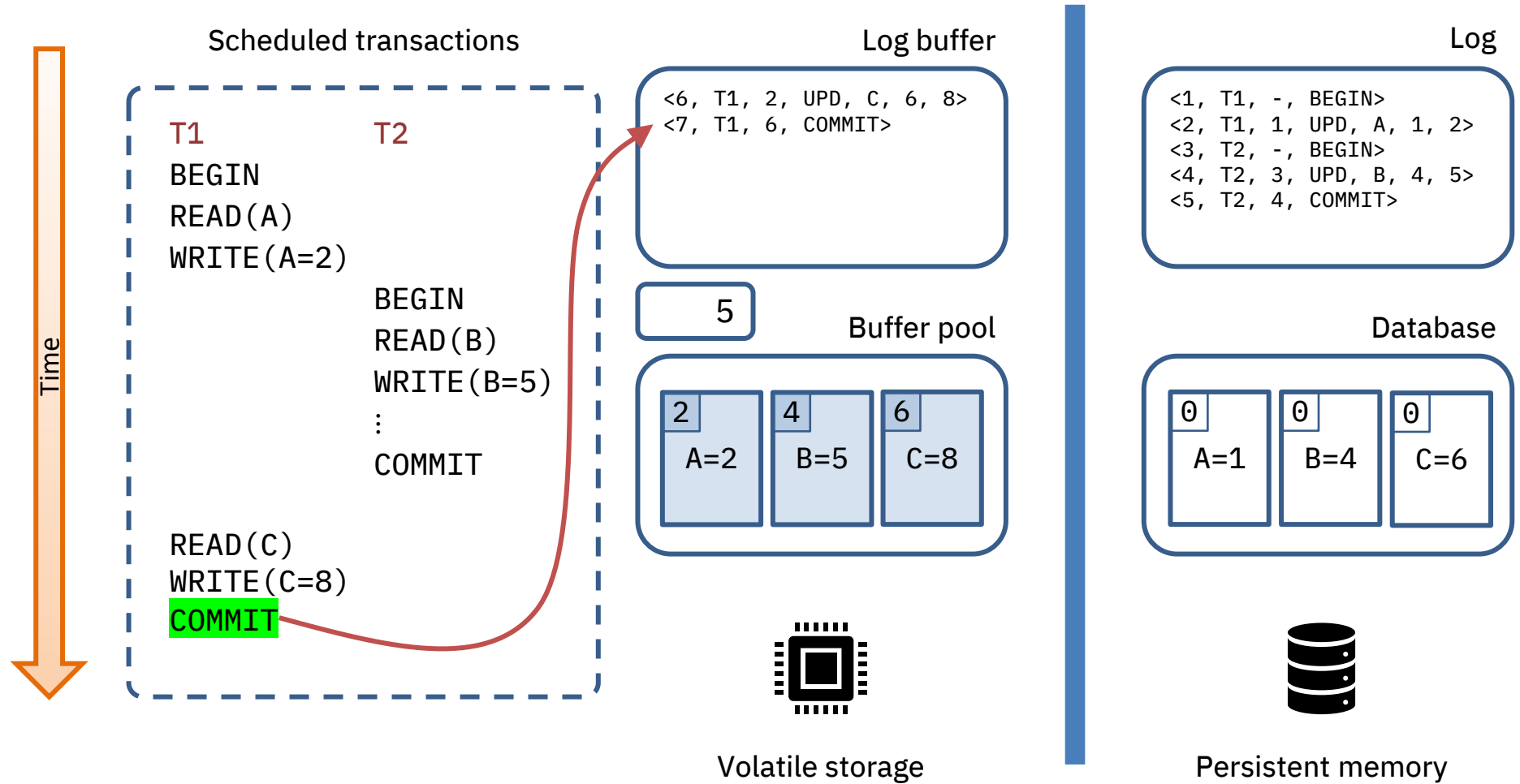


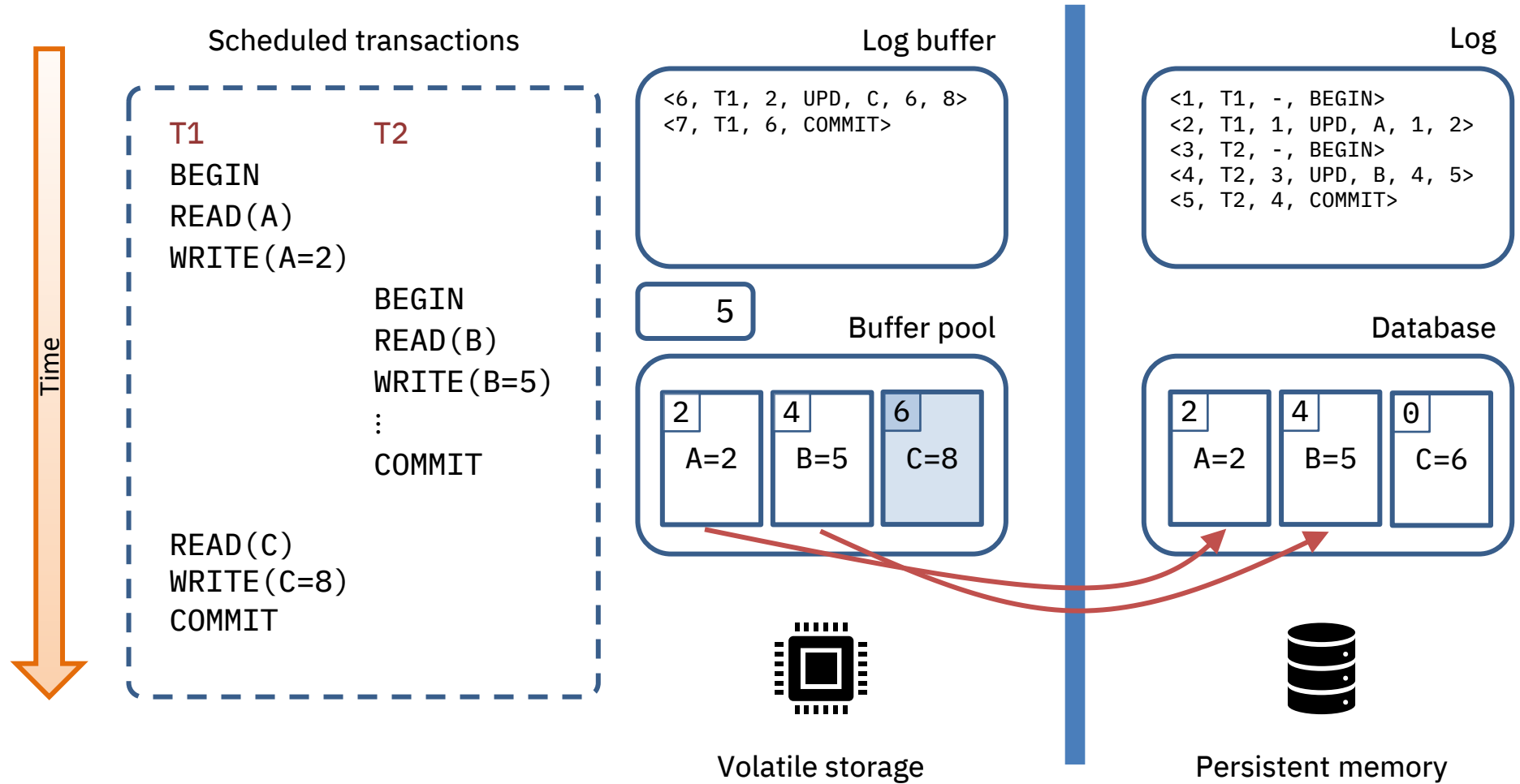




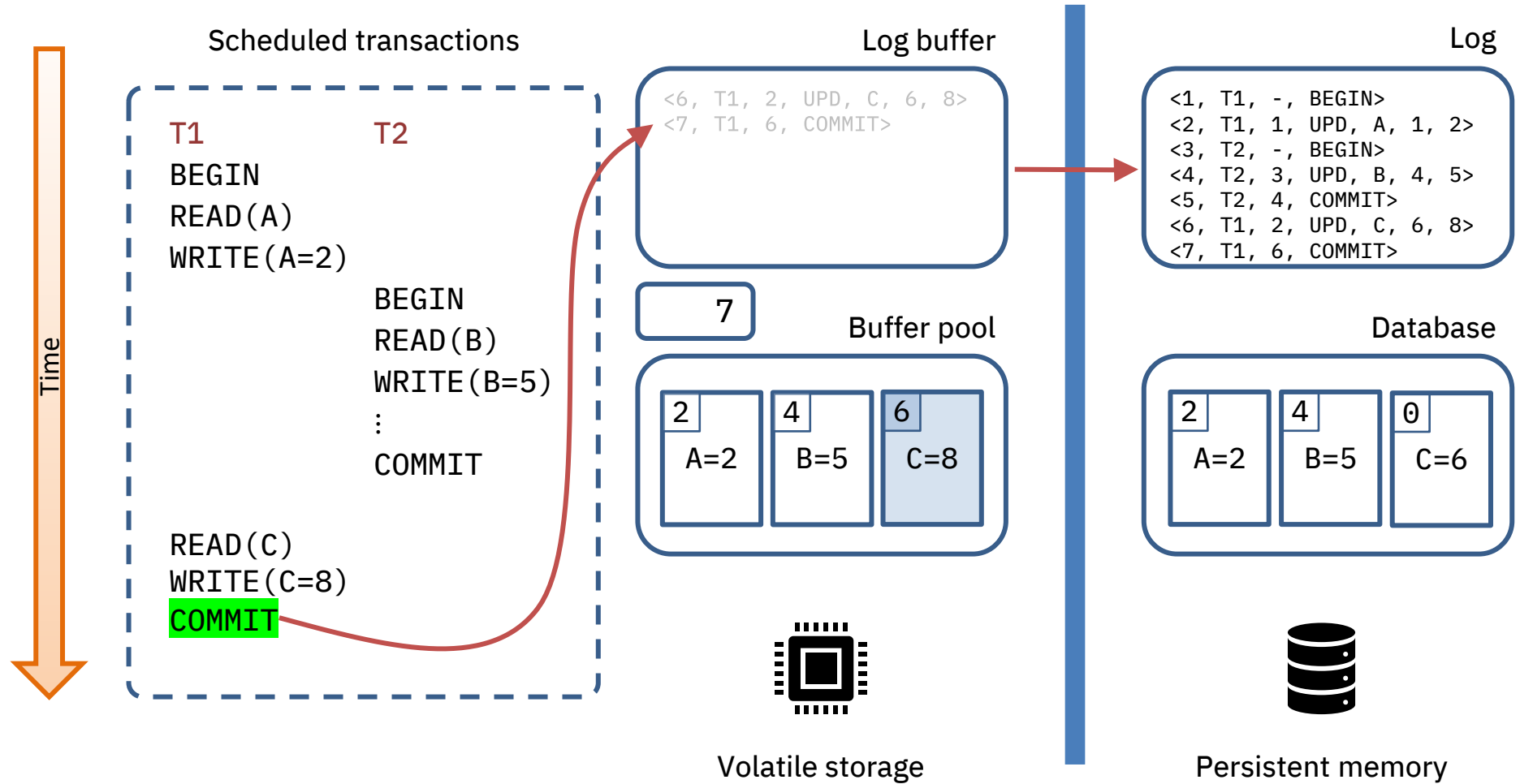


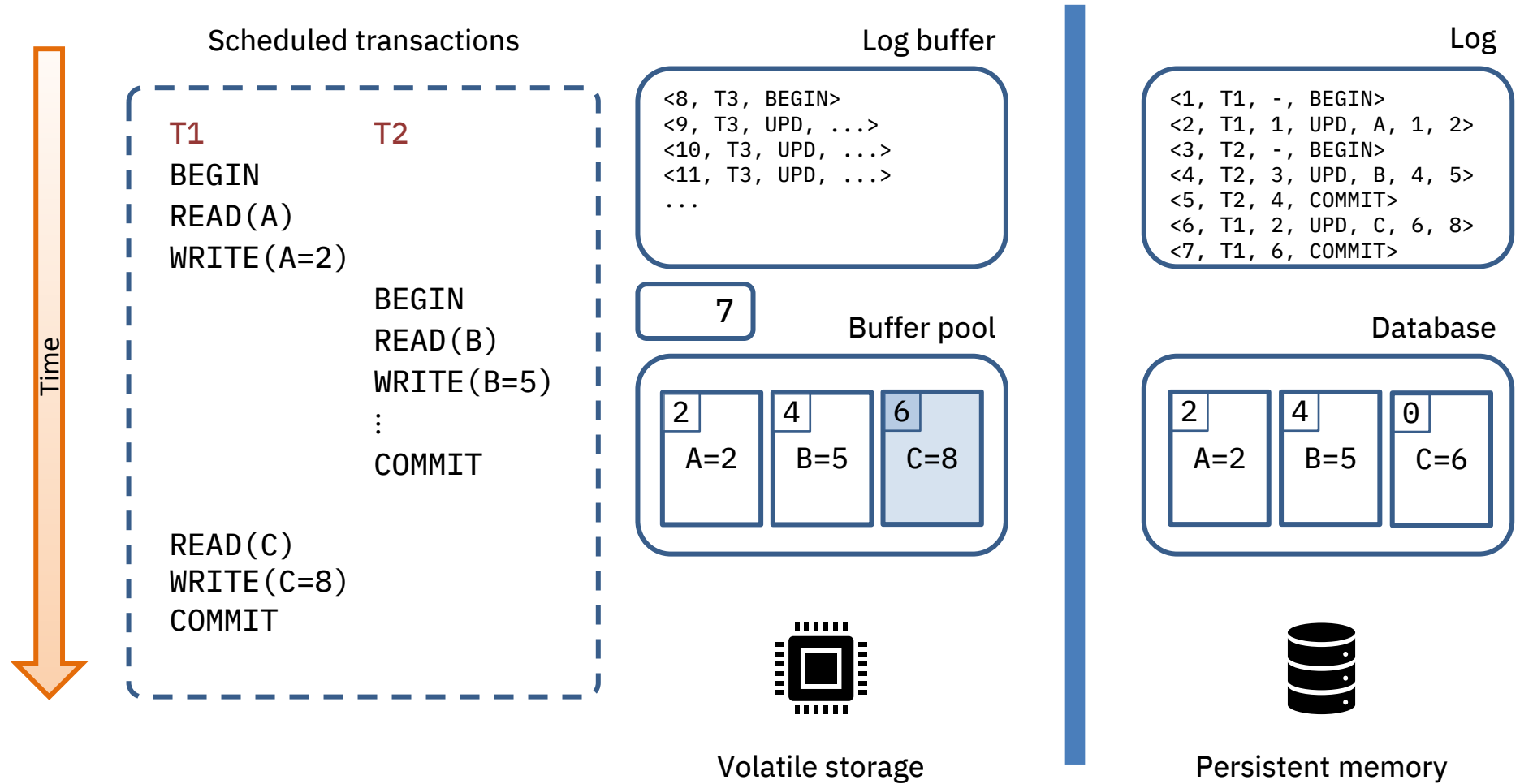














# Protokollierung von Datenänderungen

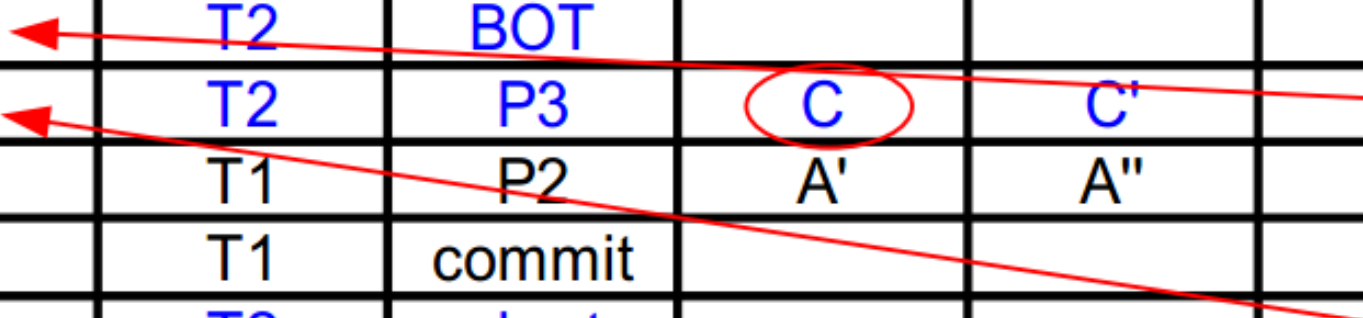
## Beispiel einer Log-Datei

<i>LSN</i>	<i>TA</i>	<i>PageID</i>	<i>Undo</i>	<i>Redo</i>	<i>PrevLSN</i>
1	T1	BOT			0
2	T1	P2	A	A'	1
3	T2	BOT			0
4	T2	P3	B	B'	3
5	T1	P2	A'	A''	2
6	T1	commit			5

# Transaktionsfehler (R1-Recovery)

## Log-Datei

<i>LSN</i>	<i>TA</i>	<i>PageID</i>	<i>Undo</i>	<i>Redo</i>	<i>PrevLSN</i>
1	T1	BOT			0
2	T1	P2	A	A'	1
3	T2	BOT			0
4	T2	P3	C	C'	3
5	T1	P2	A'	A''	2
6	T1	commit			5
7	T2	abort			4



- Ausführung des Log-Files in umgekehrter Reihenfolge
- Einbringen der Undo-Informationen



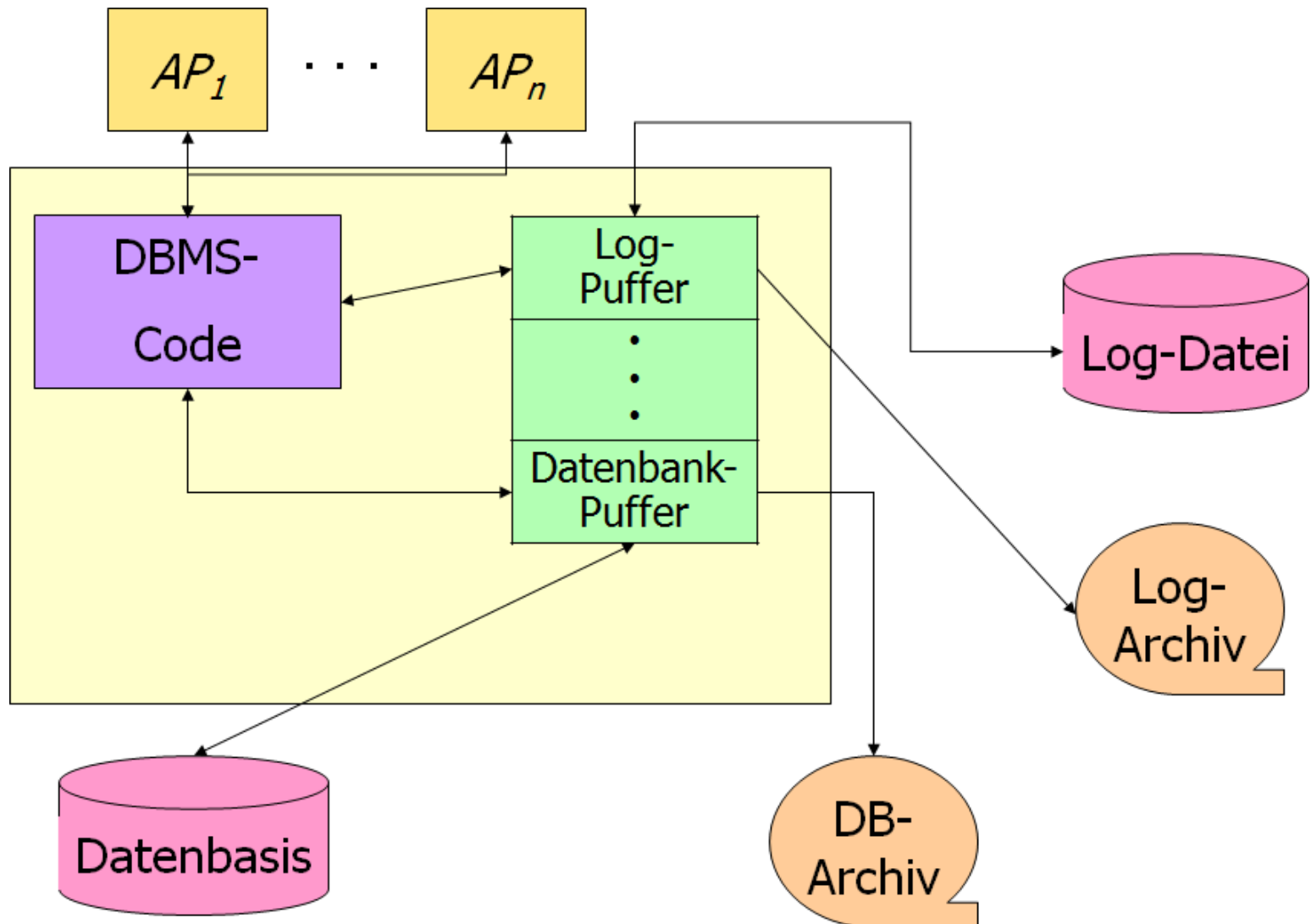
# Systemfehler

## Übersicht

- Typisch sind
  - Defekte Hardware (RAM, Prozessor, ...), DBMS-Fehler (Überlauf, sonstiger Absturz), OS-Fehler
- Wirkung?
  - Informationen im RAM sind verloren
  - Persistente Informationen auf der Platte jedoch noch vorhanden, wobei einige Daten ggf. nur teilweise weggeschrieben werden konnten
- Vorgehen für das Recovery
  - Beendete Transaktionen mit REDO nachvollziehen (R2-Recovery)
  - UNDO für nicht beendete Transaktionen (R3-Recovery)

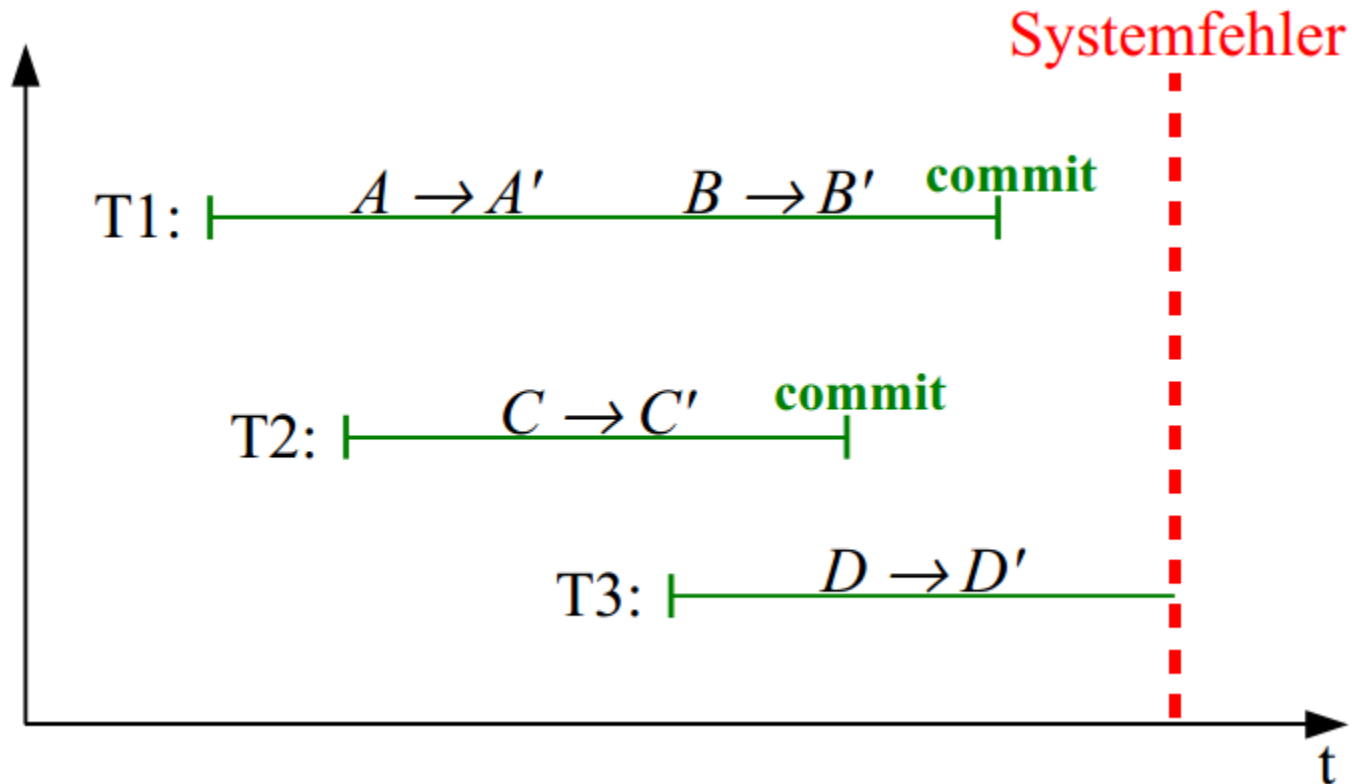
# Log-File Architektur

## grobe Übersicht



# Systemfehler (R2-, R3-Recovery)

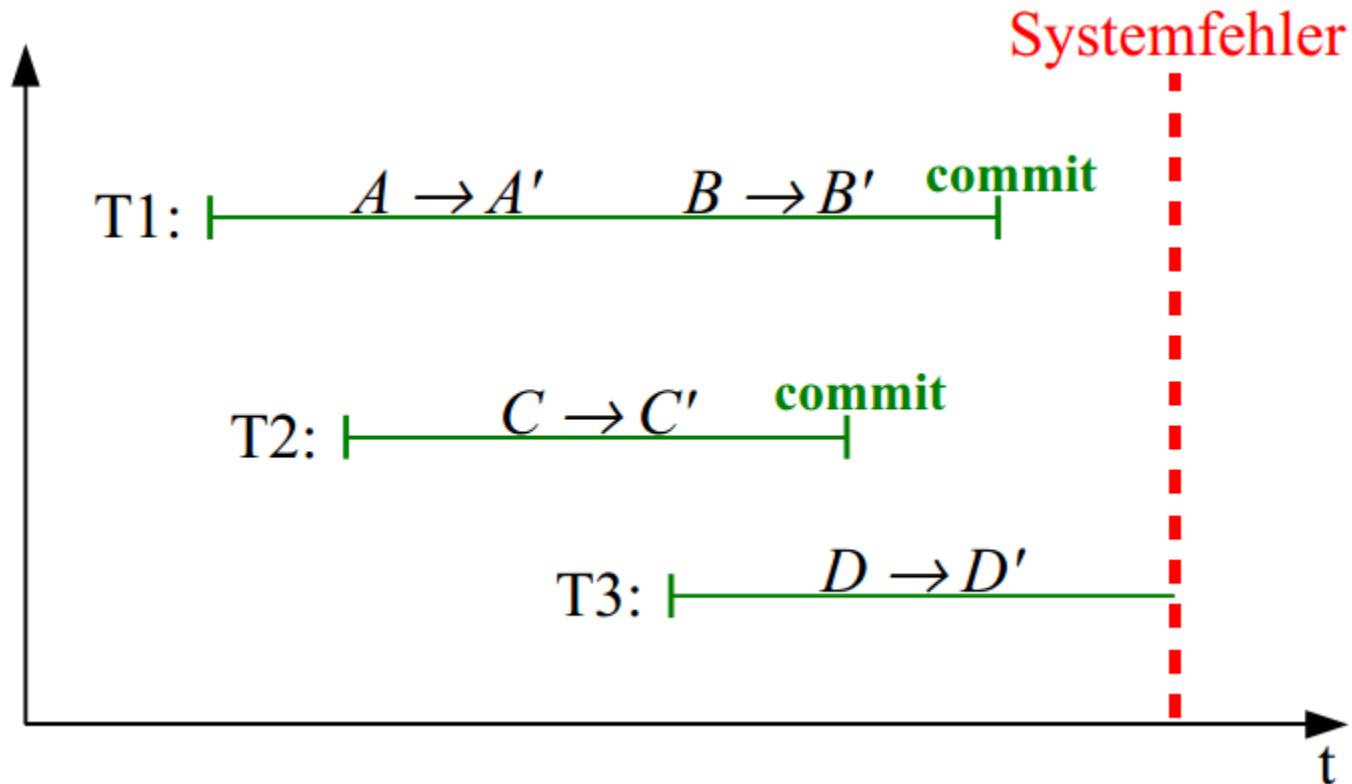
## Szenario



- T3 muss behandelt werden, als wäre sie mit abort abgebrochen worden, für T1 und T2 muss nach Neustart sichergestellt werden, dass diese im persistenten Speicher liegen

# Systemfehler (R2-, R3-Recovery)

## Szenario, Zustände in persistenter Datenbank



- T1 (committed, winner):  $A'$  wurde bereits zurück geschrieben;  $B'$  nicht(!)
- T2 (committed, winner):  $C'$  wurde bereits zurück geschrieben
- T3 (offen, loser):  $D'$  wurde bereits zurück geschrieben(!)





# Systemfehler (R2-, R3-Recovery)

## Wie realisiert man nun das Recovery?

- Phase 1: Analyse
  - Die temporäre Log-Datei wird von Anfang bis zum Ende analysiert
  - (Ermittlung der Winner-Menge von Transaktionen des Typs T1, T2)
  - Ermittlung der Loser-Menge von Transaktionen der Art T3
- Phase 2: Wiederholung der Historie
  - *alle* protokollierten Änderungen werden in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht.
- Phase 3: Undo der Loser
  - Änderungsoperationen der zum Fehlerzeitpunkt offenen (loser-) Transaktionen rückgängig machen



# Systemfehler (R2-, R3-Recovery)

## WAL: Write-Ahead-Log

- Bevor eine Transaktion festgeschrieben (committed) wird, müssen alle ihre Log-Einträge ins Log ausgeschrieben worden sein
  - → REDO im Fehlerfall
  - Demnach ist eine **Transaktion dann committed, wenn dies im Log steht** und nicht, wenn alles auf der Festplatte angekommen ist
- Bevor eine modifizierte Seite aus dem Puffer zurückgeschrieben werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in das Log geschrieben worden sein
  - → UNDO im Fehlerfall



# Große Log-Files

## Checkpoints

- Problem: wenn auf der DB viele Operationen ausgeführt wurden, ist das im Fehlerfall zu verarbeitende Log-File sehr groß
- Lösung: Checkpoints
  - bei einem Checkpoint werden alle veränderten Datenbankseiten aus dem Puffer in die Datenbank geschrieben und dies im Log vermerkt
  - Log muss im Fehlerfall nur ab dem letzten Checkpoint analysiert werden
- Häufige Checkpoints mindern Performance, reduziert aber Wiederherstellungszeit nach Fehlerfall enorm
  - Fragestellungen: alle veränderten Seiten im Puffer? Wie lange dauert das? Dürfen da andere Prozesse laufen? Es gibt unterschiedliche Qualitäten der



# Externspeicherfehler (R4-Recovery)

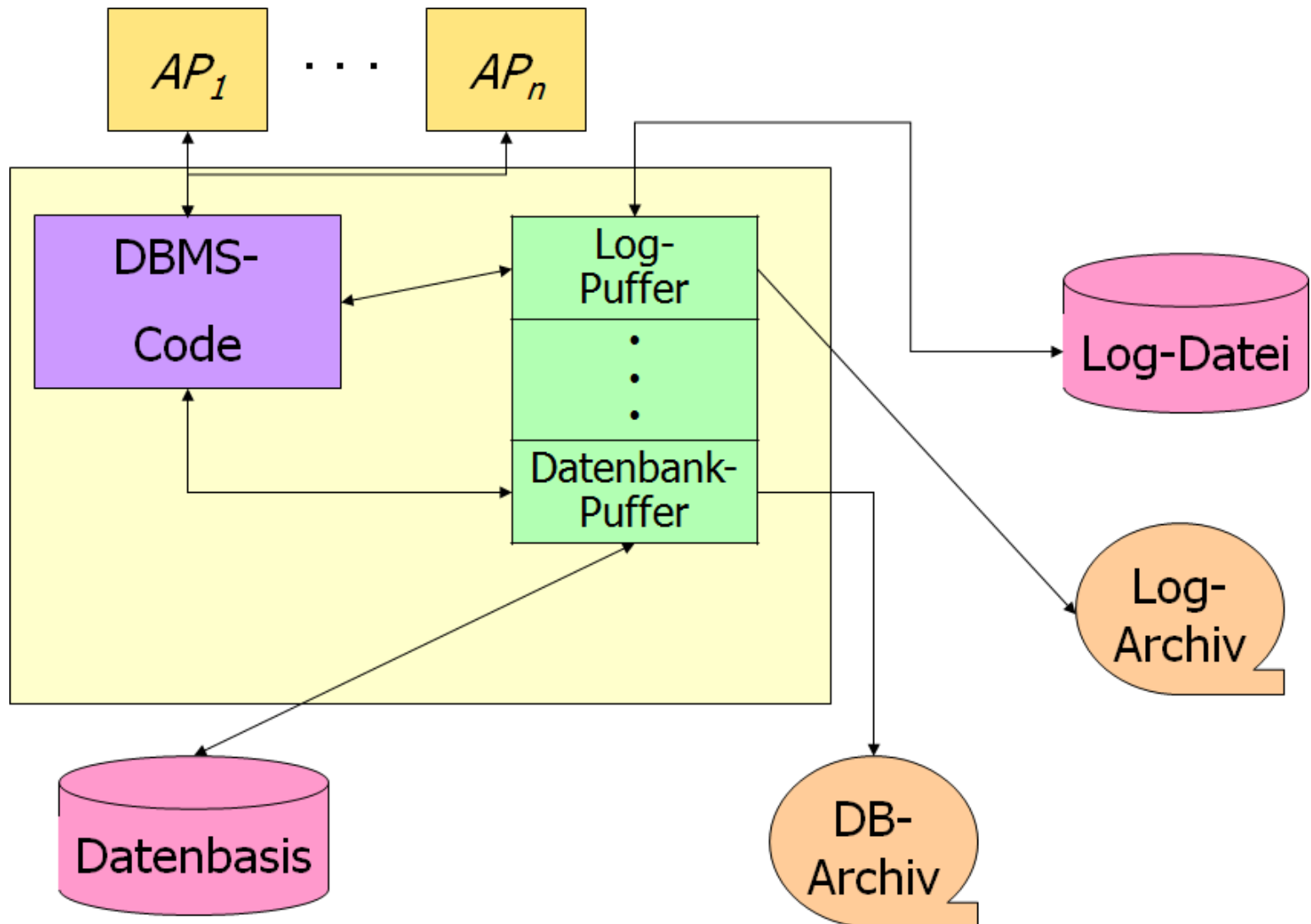
## Übersicht

- Typisch sind
  - Defekte Hardware (Festplattenkopf, ...)
  - Hacker, Viren, Würmer, ...
  - Naturgewalten
- Wirkung?
  - Persistente Daten sind zerstört / unbrauchbar
- Vorgehen für das Recovery
  - Mit Backup wiederherstellen
  - Alle seit Erstellung des Backups erfolgreich abgeschlossenen TA wiederherstellen → Log-Archiv auslagern / spiegeln



# Log-File Architektur

## grobe Übersicht





# Externspeicherfehler (R4-Recovery)

## Vorgehen

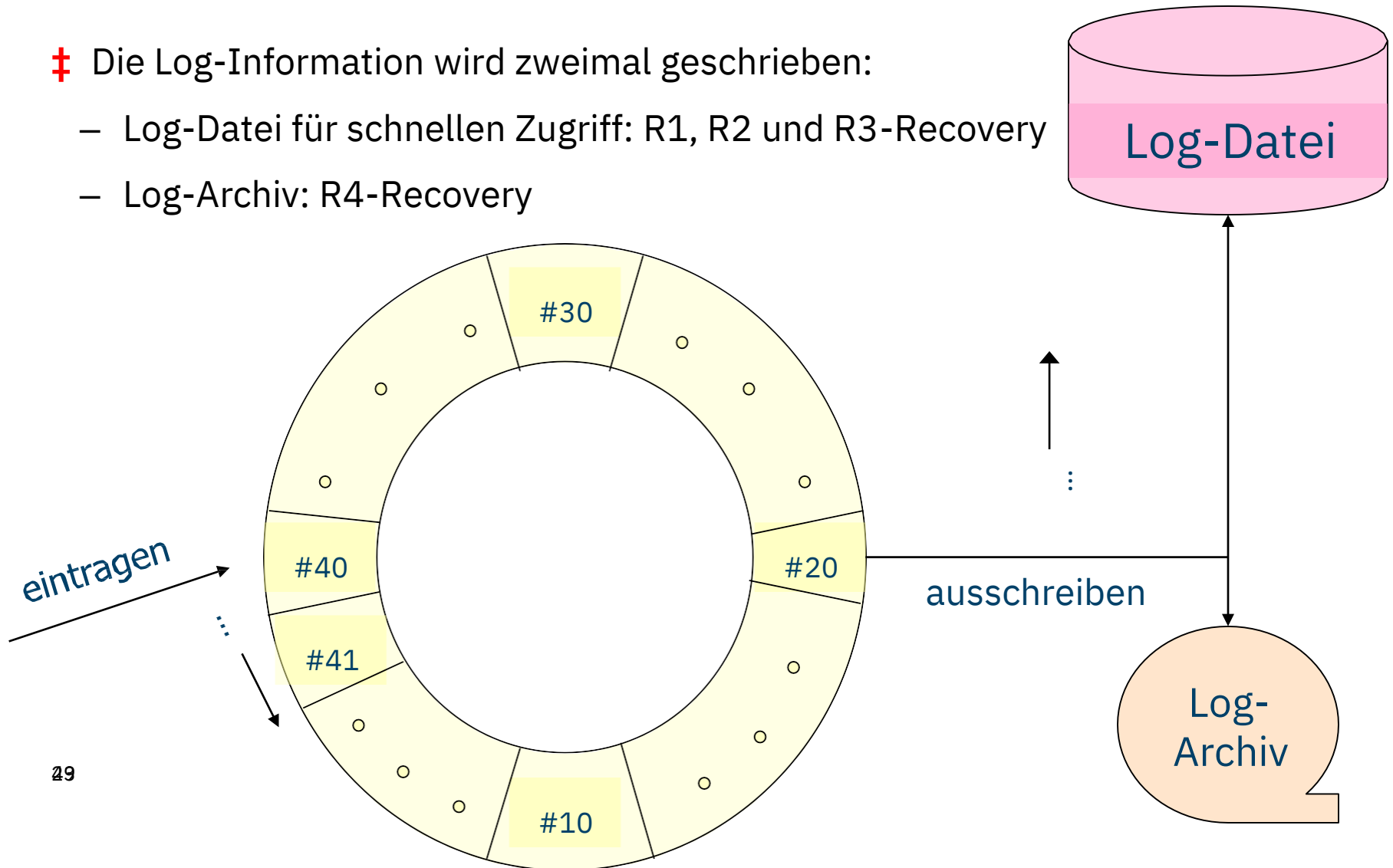
- DB-Archiv muss existieren → Wie wurde dies erstellt?
  - Offline-Backup (konsistent)
    - weniger aufwand beim Recovery
    - DBMS darf beim Backup keine TA verarbeiten
  - Online-Backup (inkonsistent)
    - mehr Aufwand beim Recovery
    - DBMS darf auch beim Backup TA verarbeiten
- Full-Backup sehr aufwendig → inkrementelles Backup die Regel



# Log-Infos

## Redundant

- ‡ Die Log-Information wird zweimal geschrieben:
- Log-Datei für schnellen Zugriff: R1, R2 und R3-Recovery
  - Log-Archiv: R4-Recovery





# Mehrbenutzersynchronisation

## Einführung

- Bis jetzt gingen wir immer von TA aus, die sich gegenseitig nicht beeinflussen
- Was passiert wenn 2 TA auf die gleiche Datensätze lesen und schreibend oder schreibend und schreibend zugreifen wollen?
  - Folge sind verschiedene Anomalien
  - Mechanismen für Synchronisation müssen umgesetzt werden

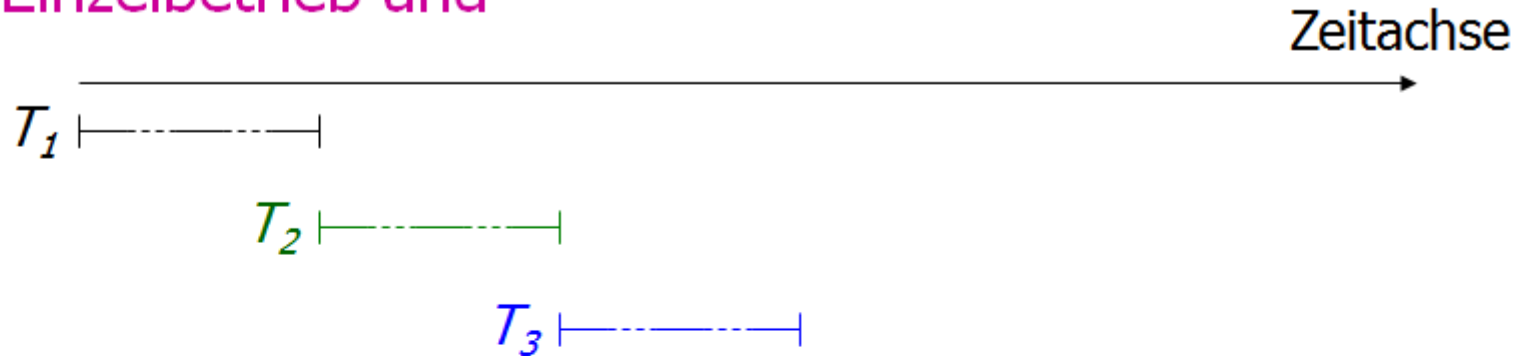




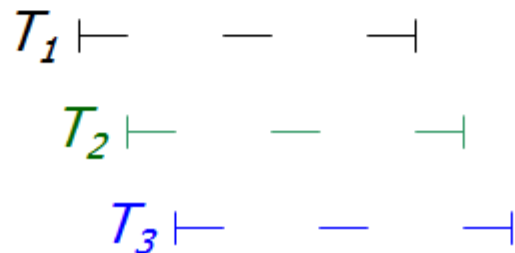
# Mehrbenutzersynchronisation

## Ausführung der drei Transaktionen $T_1$ , $T_2$ und $T_3$ :

(a) im Einzelbetrieb und



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





# Mehrbenutzersynchronisation

## Motivation

- Serielle Ausführung von TA nicht erwünscht
  - Wartezeiten
- Parallele Nutzung liefert i. A. bessere Auslastung des Systems
  - Wenn auf I/O gewartet wird, kann eine andere TA ggf. zum großen Teil aus dem Puffer bearbeitet werden
- „Logischer“ 1-Benutzer-Betrieb soll aber trotzdem gewährleistet werden
  - Was passiert, wenn dies nicht korrekt geschieht? Folgefolien!

# Mehrbenutzersynchronisation - Anomalien

## Lost Update

- Änderungen einer Transaktion können durch Änderungen anderer Transaktionen 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“;

- Mgl. 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



# Mehrbenutzersynchronisation - Anomalien

## Dirty Read (Dirty Write)

- Zugriff auf „schmutzige“ Daten, d.h. auf Objekte, die von einer noch nicht abgeschlossenen Transaktion geändert wurden
- Bsp.:
  - T1 erhöht das Gepäck um 3 kg, wird aber später abgebrochen
  - T2 erhöht das Gepäck um 5 kg und wird erfolgreich abgeschlossen
- Mgl. Ablauf:

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

- Durch den Abbruch von T1 werden die geänderten Werte ungültig. T2 hat jedoch die geänderten Werte gelesen (*Dirty Read*) und weitere Änderungen darauf aufgesetzt (*Dirty Write*)
- Verstoß gegen ACID: Dieser Ablauf verursacht einen inkonsistenten DB-Zustand (*Consistency*) bzw. T2 muss zurückgesetzt werden (*Durability*).

# Mehrbenutzersynchronisation - Anomalien

## Non-Repeatable Read

- Eine Transaktion sieht während ihrer Ausführung unterschiedliche Werte desselben Objekts
- Bsp.:
  - 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
- Mgl. Ablauf

T1	T2
<pre>SELECT Gepäck FROM Passagiere WHERE FlugNr = „BA932“;  SELECT Gepäck FROM Passagiere WHERE FlugNr = „BA932“;</pre>	<pre>UPDATE Passagiere SET Gepäck = Gepäck+5; COMMIT;</pre>

- Die beiden SELECT-Anweisungen von Transaktion T1 liefern unterschiedliche Ergebnisse, obwohl die T1 den DB-Zustand nicht geändert hat → Verstoß gegen *Isolation*



# Mehrbenutzersynchronisation - Anomalien

## Phantome (Sonderfall von non rep. read)

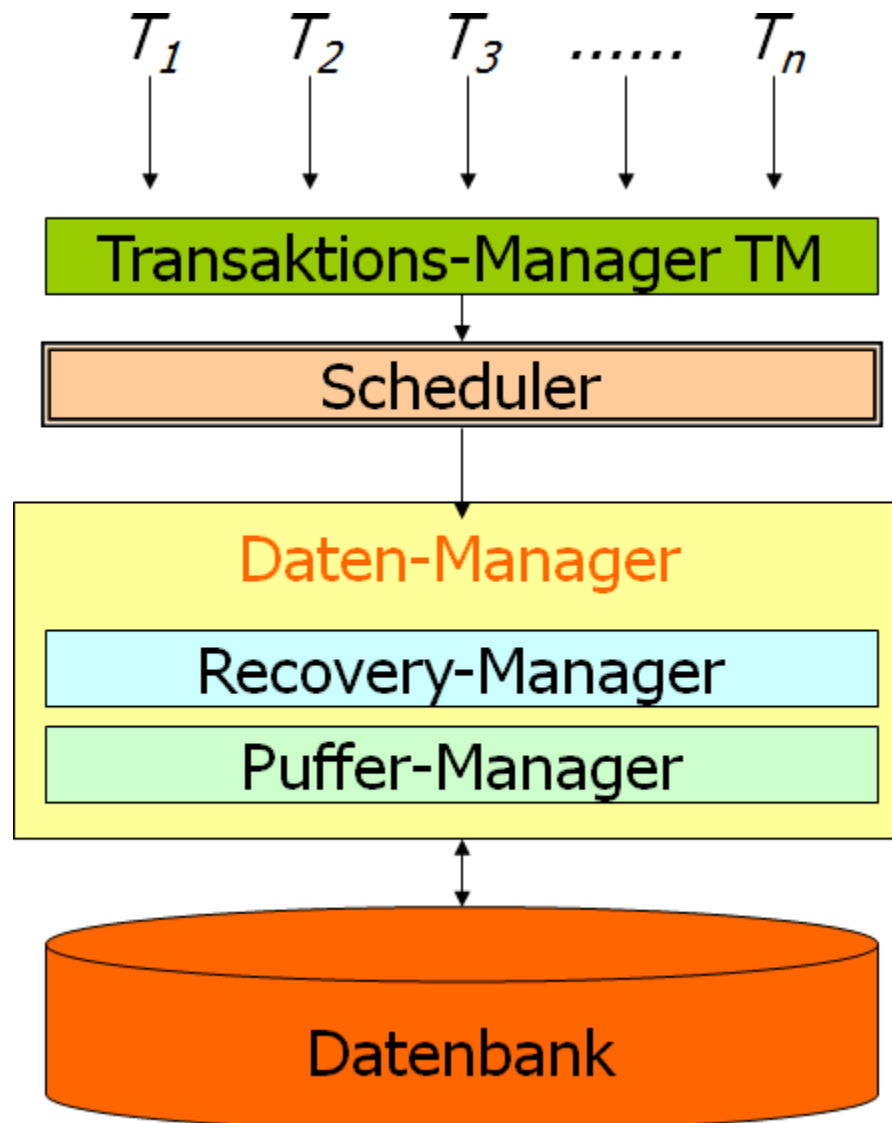
- eine Transaktion berechnet einen Wert basierend auf den aktuellen Daten, daraufhin werden neue Daten von einer weiteren Transaktion hinzugefügt und dann arbeitet die erste Transaktion weiter basierend auf den vorher ermittelten Werten

Zeit	Transaktion 1	Transaktion 2
1	<code>x := select count(*) from Personal</code>	
2		<code>insert into Personal (Pnr,Name,Gehalt) values (101,'Tom Jones',40000.0)</code>
3		<code>commit</code>
4	<code>update Personal set Gehalt=Gehalt+10000.0/x</code>	
5	<code>commit</code>	



# Mehrbenutzersynchronisation

## Scheduler





# Mehrbenutzersynchronisation

## Schedules (Historien)

- 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





# Mehrbenutzersynchronisation

## Serialisierbarkeit

- Der Scheduler kümmert sich darum, welche Aktionen welcher Transaktionen wann ausgeführt werden. Begriffe sind hier:
  - **Schedule** (auch: „Historie“) für Menge  $\{TA_1, \dots, TA_n\}$  ist Folge von Aktionen, die durch Mischen von Aktionen der beteiligten TA entsteht. Die Reihenfolge der Aktionen innerhalb jeder TA bleibt dabei unverändert.
  - Eine **Aktion** kann sein: Lesen  $r_i(A)$ , Schreiben  $w_i(A)$ , Commit  $c_i$ , Rollback  $a_i$
  - Ein **serieller Schedule** ist ein Schedule von  $\{TA_1, \dots, TA_n\}$ , in dem die Aktionen der einzelnen Transaktionen nicht untereinander verzahnt, sondern in Blöcken hintereinander ausgeführt werden
  - Ein Schedule von  $\{TA_1, \dots, TA_n\}$  ist **serialisierbar**, wenn er garantiert dieselbe Wirkung hat wie mind. 1 beliebiger serieller Schedule von  $\{TA_1, \dots, TA_n\}$

→ nur diese Schedules zulassen!

# Mehrbenutzersynchronisation

## Serialisierbarkeit

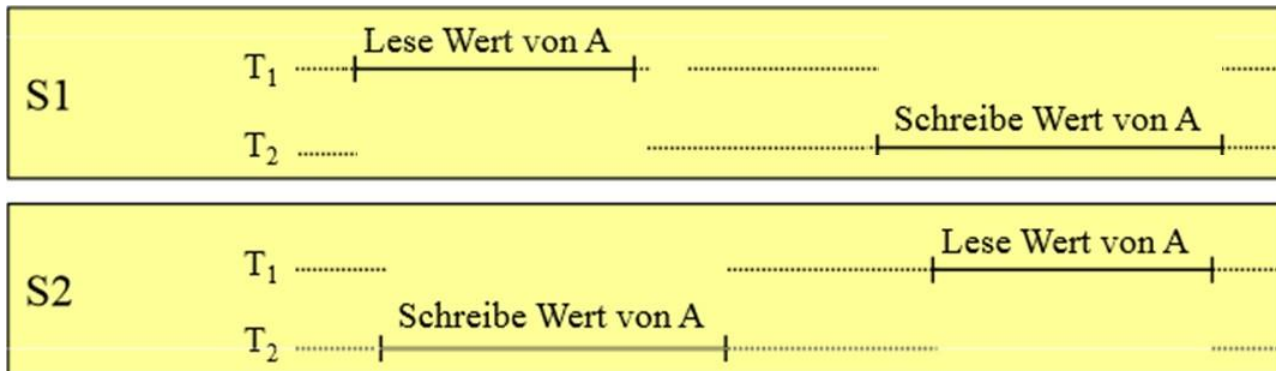
?

**Nun stellt sich die Frage, wann ein  
gegebener Schedule die gleiche  
Wirkung auf ein System hat wie ein  
anderer beliebiger Schedule... Ideen?**

# Mehrbenutzersynchronisation

## Serialisierbarkeit :: Konflikt-Äquivalenz.

- Wenn in unserem gegebenen Schedule S1 die Transaktion T1 einen Wert schreibt und T2 diesen Wert dann liest, dann muss dies auch in jedem anderen Schedule gelten



**Gibt es auch Operationen, bei denen die Reihenfolge keine Rolle spielt? Warum?**

- Es gibt also eine Abhängigkeit, bei der die Reihenfolge zwingend eingehalten werden muss – würden wir die Reihenfolge vertauschen: anderes Ergebnis.
  - Wir sprechen hier also einen einem **Konflikt**



# Mehrbenutzersynchronisation

## Serialisierbarkeit :: Konflikt-Äquivalenz.

- **Konflikte** zwischen  $TA_i$  und  $TA_j$  entstehen demzufolge durch **Abhängigkeiten**, bei denen Schreiboperationen beteiligt sind. Also wenn:
  - $w_i(x)$  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  $ww(x)$
- Zwei Schedules  $S1$  und  $S2$  sind genau dann **konfliktäquivalent**, wenn:
  - Beide Schedules die gleichen Transaktionen mit jeweils den gleichen Operationen (in gleicher Reihenfolge bezogen auf die jeweilige Transaktion) beinhalten
  - Beide Schedules gleiche Menge an Abhängigkeiten zueinander besitzen
- Wenn zwei Schedules konfliktäquivalent sind, dann haben sie dieselbe Wirkung auf den Datenbestand, also den Inhalt der Datenbank

# Mehrbenutzersynchronisation

## Serialisierbarkeit :: Konflikt-Äquivalenz, ein Beispiel

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

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

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

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

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

$A_{S_1} = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_2(x), w_1(x))\}$

$A_{S_2} = \{(r_2(x), w_1(x)), (r_1(x), w_2(x)), (w_2(x), w_1(x))\}$

$A_{S_3} = \{(r_1(x), w_2(x)), (r_2(x), w_1(x)), (w_1(x), w_2(x))\}$

- 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.



# Mehrbenutzersynchronisation

## Serialisierbarkeit :: Konflikt-Äquivalenz, ein Beispiel

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

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

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

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

- Aktionsmengen von  $S_1, S_2, S_3, S_4$  sind identisch
- Abhängigkeitsmengen:

$$A_{S_1} = \{ r_1 w_2(x), r_2 w_1(x), w_2 w_1(x) \}$$

$$A_{S_2} = \{ r_2 w_1(x), r_1 w_2(x), w_2 w_1(x) \}$$

$$A_{S_3} = \{ r_1 w_2(x), r_2 w_1(x), w_1 w_2(x) \}$$

- Schedules  $S_1$  und  $S_2$  sind konfliktäquivalent
- Schedules  $S_1$  und  $S_3$  sowie  $S_2$  und  $S_3$  sind nicht konfliktäquivalent
- Schedule  $S_4$  ist kein Schedule derselben Tx, da die Aktionen innerhalb der Tx vertauscht sind



# Mehrbenutzersynchronisation

## Beispiel: Zunächst die serielle Schedules

Schritt	T1	T2
1	<b>BOT</b>	
2	read(A)	
3	write(A)	
4	read(B)	
5	write(B)	
6	<b>commit</b>	
7		<b>BOT</b>
8		read(A)
9		write(A)
10		read(B)
11		write(B)
12		<b>commit</b>

Schritt	T1	T2
1	<b>BOT</b>	
2	read(A)	
3	write(A)	
4		<b>BOT</b>
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		<b>commit</b>
10	read(B)	
11	write(B)	
12	<b>commit</b>	



# Mehrbenutzersynchronisation

## Beispiel: Zwei Überweisungen, seriell

	T1		T2		A	B
1	<b>BOT</b>				500	200
2	read(A)	update konto set btr = btr - 50 where ktonr = A				
3	write(A)				450	200
4	read(B)	update konto set btr = btr + 50 where ktonr = B				
5	write(B)				450	250
6	<b>commit</b>					
7			<b>BOT</b>			
8			read(A)	update konto set btr = btr - 100 where ktonr = A		
9			write(A)		350	250
10			read(B)	update konto set btr = btr + 100 where ktonr = B		
11			write(B)		350	350
12			<b>commit</b>			





# Mehrbenutzersynchronisation

**Beispiel: Überweisungen, 2% Zinsgutschriften, seriell**

	T1		T2		A	B
1	BOT				500	200
2	read(A)	update konto set btr = btr - 50 where ktonr = A				
3	write(A)				450	200
4	read(B)	update konto set btr = btr + 50 where ktonr = B				
5	write(B)				450	250
6	commit					
7			BOT			
8			read(A)	update konto set btr = btr * 1.02 where ktonr = A		
9			write(A)		459	250
10			read(B)	update konto set btr = btr * 1.02 where ktonr = B		
11			write(B)		459	255
12			commit			



# Mehrbenutzersynchronisation

**Beispiel: Überweisungen, 2% Zinsgutschriften, seriell**

	T1		T2		A	B
1			BOT		500	200
2			read(A)	update konto set btr = btr * 1.02 where ktonr = A		
3			write(A)		510	200
4			read(B)	update konto set btr = btr * 1.02 where ktonr = B		
5			write(B)		510	204
6			commit			
7	BOT				510	204
8	read(A)	update konto set btr = btr - 50 where ktonr = A				
9	write(A)				460	204
10	read(B)	update konto set btr = btr + 50 where ktonr = B				
11	write(B)				460	254
12	commit					



# Mehrbenutzersynchronisation

## Beispiel: Jetzt die verzahnten Schedules

Schritt	T1	T2
1	<b>BOT</b>	
2	read(A)	
3	write(A)	
4	read(B)	
5	write(B)	
6	<b>commit</b>	
7		<b>BOT</b>
8		read(A)
9		write(A)
10		read(B)
11		write(B)
12		<b>commit</b>

Schritt	T1	T2
1	<b>BOT</b>	
2	read(A)	
3	write(A)	
4		<b>BOT</b>
5		read(A)
6		write(A)
7		read(B)
8		write(B)
9		<b>commit</b>
10	read(B)	
11	write(B)	
12	<b>commit</b>	



# Mehrbenutzersynchronisation

## Beispiel: Überweisungen, verzahnt

	T1		T2		A	B
1	BOT				500	200
2	read(A)	update konto set btr = btr - 50 where ktonr = A				
3	write(A)				450	200
4			BOT			
5			read(A)	update konto set btr = btr - 100 where ktonr = A	350	
6			write(A)			
7			read(B)	update konto set btr = btr + 100 where ktonr = B		300
8			write(B)			
9			commit			
10	read(B)	update konto set btr = btr + 50 where ktonr = B				
11	write(B)				350	350
12	commit					



# Mehrbenutzersynchronisation

## Beispiel: Überweisungen, 2% Zinsgutschriften, verzahnt

	T1		T2		A	B
1	BOT				500	200
2	read(A)	update konto set btr = btr - 50 where ktonr = A				
3	write(A)				450	200
4			BOT			
5			read(A)	update konto set btr = btr * 1.02 where ktonr = A	459	
6			write(A)			
7			read(B)	update konto set btr = btr * 1.02 where ktonr = B		204
8			write(B)			
9			commit			
10	read(B)	update konto set btr = btr + 50 where ktonr = B				
11	write(B)				459	254
12	commit					



# Mehrbenutzersynchronisation

## Konfliktgraphen / Serialisierungsgraphen

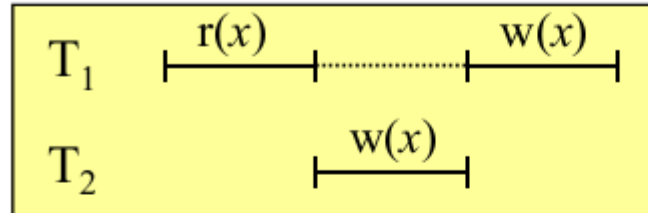
- Mit Hilfe von Konfliktgraphen / Serialisierungsgraphen (gerichtet, irreflexiv) kann man prüfen, ob ein Schedule  $\{TA_1, \dots, TA_n\}$  serialisierbar ist
- Jede TA ist Knoten im Graph, jede Kante zwischen den Knoten bezeichnet eine Abhängigkeit. Eine gerichtete Kante  $TA_i \rightarrow TA_j$  wird eingetragen, wenn  $i$  ungleich  $j$  und
  - $w_i(x)$  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  $ww(x)$
- Ein Schedule ist serialisierbar, wenn er **zyklenfrei** ist
  - Der konfliktäquivalente serielle Schedule kann dann für jede Zusammenhangskomponente durch **topologische Sortierung** ermittelt werden



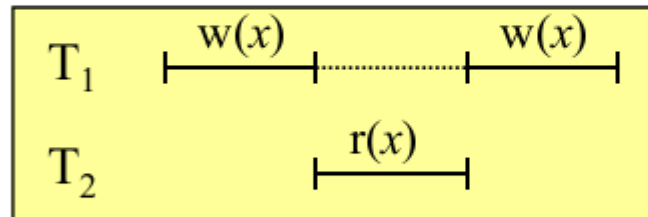
# Mehrbenutzersynchronisation

## nicht serialisierbare Schedules, Beispiele

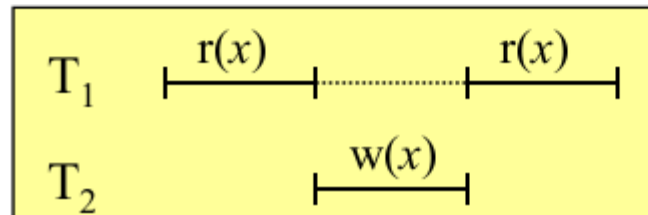
- Lost Update:  $S=(r_1(x), w_2(x), w_1(x))$



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



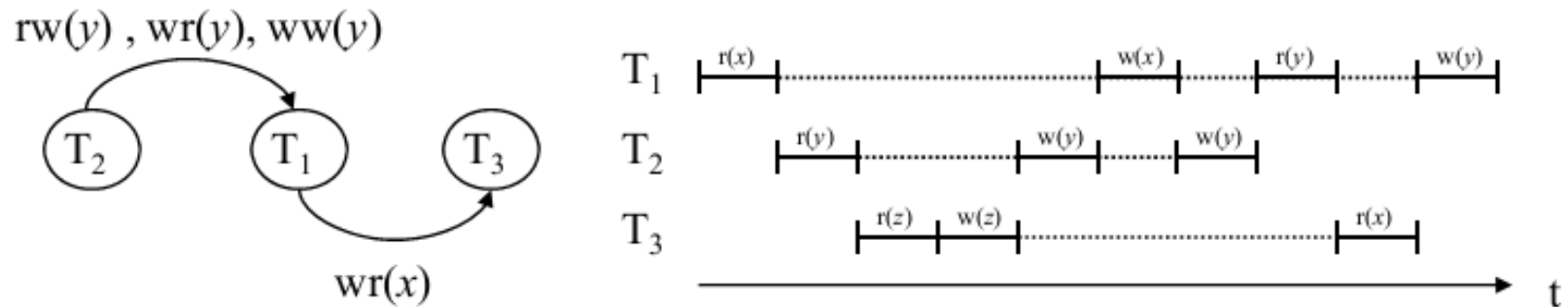
- Non-repeatable Read:  $S=(r_1(x), w_2(x), r_1(x))$



# Mehrbenutzersynchronisation

## Konfliktgraphen / Serialisierungsgraphen :: BEISPIEL

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

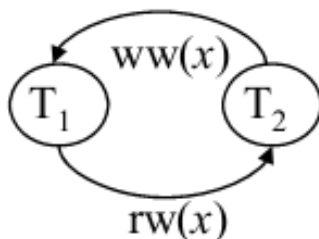


Serialisierungsreihenfolge:  $(T_2, T_1, T_3)$

- Nicht-serialisierbare Schedules

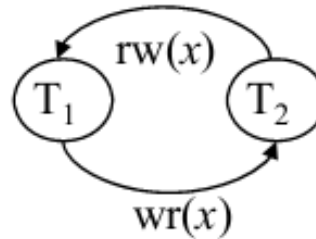
$S = (r_1(x), w_2(x), w_1(x))$

Lost Update



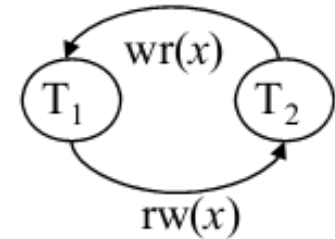
$S = (w_1(x), r_2(x), w_1(x))$

Dirty Read



$S = (r_1(x), w_2(x), r_1(x))$

Non-Repeatable Read







# Mehrbenutzersynchronisation

## Konfliktgraphen / Serialisierungsgraphen :: ÜBUNG

- Ermitteln Sie, ob folgende Schedules serialisierbar sind.
  - Falls ja, geben Sie alle möglichen Serialisierungsreihenfolgen an
  - Falls nein, nennen Sie mindestens eine mögliche Anomalie im Schedule!

$S_1 = ( r_1(x), w_2(x), w_2(z), w_1(y), w_1(z), r_3(y) )$

$S_2 = ( r_1(x), w_2(x), w_2(z), w_1(y), w_3(z), r_3(y) )$

$S_3 = ( r_1(x), w_2(x), r_1(y), w_3(z), r_3(y), w_2(z) )$



# Mehrbenutzersynchronisation

## Konfliktgraphen / Serialisierungsgraphen :: LÖSUNG

- Ermitteln Sie, ob folgende Schedules serialisierbar sind.
  - Falls ja, geben Sie alle möglichen Serialisierungsreihenfolgen an
  - Falls nein, nennen Sie mindestens eine mögliche Anomalie im Schedule!

$S_1 = ( r_1(x), w_2(x), w_2(z), w_1(y), w_1(z), r_3(y) )$

→ Lost Update Anomalie zwischen den ersten 2 TA

$S_2 = ( r_1(x), w_2(x), w_2(z), w_1(y), w_3(z), r_3(y) )$

→ serialisierbar: T1, T2, T3

$S_3 = ( r_1(x), w_2(x), r_1(y), w_3(z), r_3(y), w_2(z) )$

→ serialisierbar: T1, T3, T2 sowie T3, T1, T2



# Mehrbenutzersynchronisation

## Konfliktgraphen / Serialisierungsgraphen :: Zusammenfassung

- Mit Hilfe von Konfliktgraphen / Serialisierungsgraphen kann man prüfen, ob ein Schedule  $\{TA_1, \dots, TA_n\}$  serialisierbar ist
  - Ist der Fall, wenn zyklensfrei
- Ein Schedule aus nur lesenden TA: unkritisch
- ABER: kennt man vorher schon alle Befehle innerhalb einer / jeder TA?
  - NEIN, in der Regel nicht! ☹
  - Demnach könnten Konfliktgraphen in der Praxis maximal dazu genutzt werden, um herauszufinden, ob laufende TA bis zum aktuellen Zeitpunkt serialisierbar wären ... a priori schwierig, unmöglich, praxisfern!
- Manchmal verzichtet man darum sogar auf Serialisierbarkeit und läuft Gefahr, mit Anomalien konfrontiert zu werden – mehr dazu später.



# Eigenschaften von Historien

## Rücksetzbare Historien

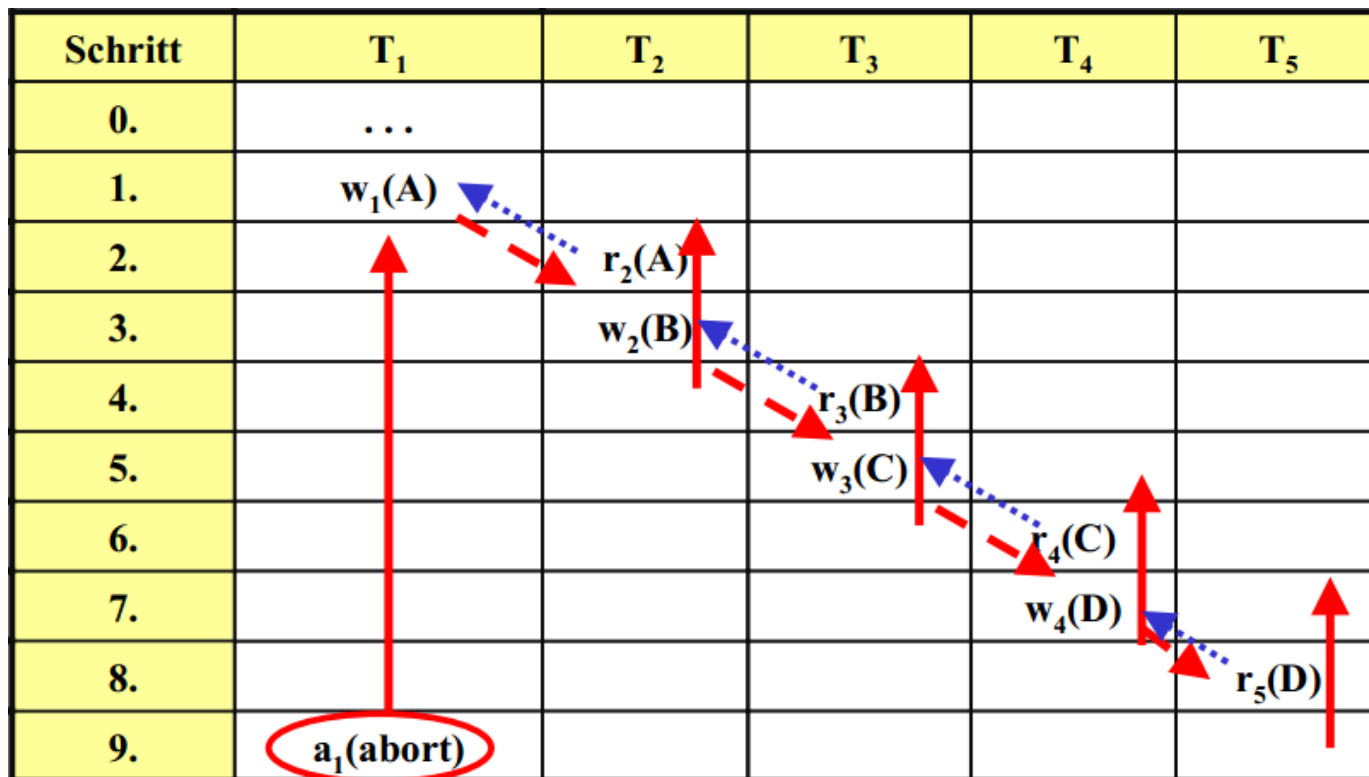
- Bisher können wir serialisierbare Historien bestimmen – super. Aber was passiert, wenn es einen Fehler gibt, alle Operationen zwar seriell ablaufen, aber eine TA zurückgerollt wird? Hier kommen „rollback“ und „commit“ ins Spiel:
- Eine Historie heißt **rücksetzbar**, falls immer die schreibende Transaktion  $T_w$  vor der lesenden Transaktion  $T_r$  ihr commit durchführt, also:  $C_w <_H C_r$ .
- Anders ausgedrückt: Eine Transaktion darf erst dann ihr commit durchführen, wenn alle Transaktionen, von denen sie gelesen hat, beendet sind. Sonst:

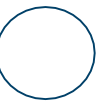
$T_w$	write(A,\$x)	abort
$T_r$	read(A,\$x)	write(A,\$x+1)    commit

# Eigenschaften von Historien

## Historien ohne kaskadierendes Zurücksetzen

- Eine Historie vermeidet kaskadierendes Rücksetzen, wenn für je zwei TAs  $T_r$  und  $T_w$  gilt:  $c_w <_H r_r(A)$ , wann immer  $T_r$  ein Datum A von  $T_w$  liest.
- Im Klartext: Eine TA darf keine uncommitteten Änderungen lesen





# Eigenschaften von Historien

## strikte Historien

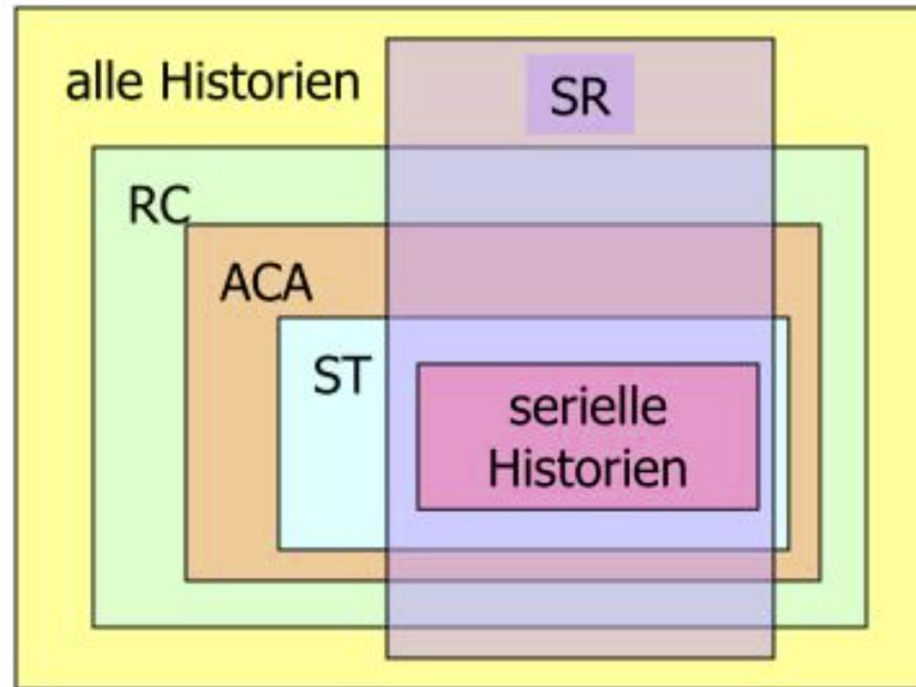
- Eine Historie ist **strikt**, wenn bei zwei Operationen  $w_1(A) <_H o_2(A)$  gilt:  
 $c_1 <_H o_2(A)$  oder  $a_1 <_H o_2(A)$  (o steht für read oder write: „operation“)
- Im Klartext: Geänderte Daten einer nicht beendeten TA dürfen durch andere TA weder gelesen noch geschrieben werden
- Wozu diese Einschränkung? Physische Protokollierung ist nur hier ohne Weiteres möglich:

```
x = 0
w1[x, 1]  before image von T1: 0
x = 1
w2[x, 2]  before image von T2: 1
x = 2
a1
c2
```



# Eigenschaften von Historien

## Beziehungen zwischen den Klassen von Historien



SR: serialisierbare Historien

RC: rücksetzbare Historien

ACA: Historien ohne kaskadierendes Rücksetzen

ST: strikte Historien



# Mehrbenutzersynchronisation

## Anforderungen

- Wegen Anomalien zumindest „serialisierbare Historien“
- Wegen ACID-Prinzip „rücksetzbare Historien“
  - Weil: nach commit nicht mehr rücksetzbar
- „Rollback-Lawine“ beim kaskadierenden Rücksetzen ist unschön, daher wollen wir keine solchen Historien
- Wenn man auf sehr einfache Art und Weise physisch protokolliert, muss ggf. sogar eine strikte Historie vorliegen





# Mehrbenutzersynchronisation

## Schedule-Klassen, Zusammenfassung

- **Seriell** (Transaktionen unverzahnt, in einzelnen Blöcken, nacheinander)
- **Konfliktäquivalent** (2 Schedules haben gleiche Wirkung auf Datenbank)
- **Serialisierbar** (Konfliktäquivalent zu einem seriellen Schedule)
- **Rücksetzbar** (Commit erst, wenn gelesene Daten auch „sicher“ sind)
- **Ohne Kaskadierendes Zurücksetzen** (veränderte Daten einer noch laufenden TA dürfen nicht gelesen werden)
- **Strikt** (veränderte Daten einer noch laufenden TA dürfen weder gelesen noch überschrieben werden)

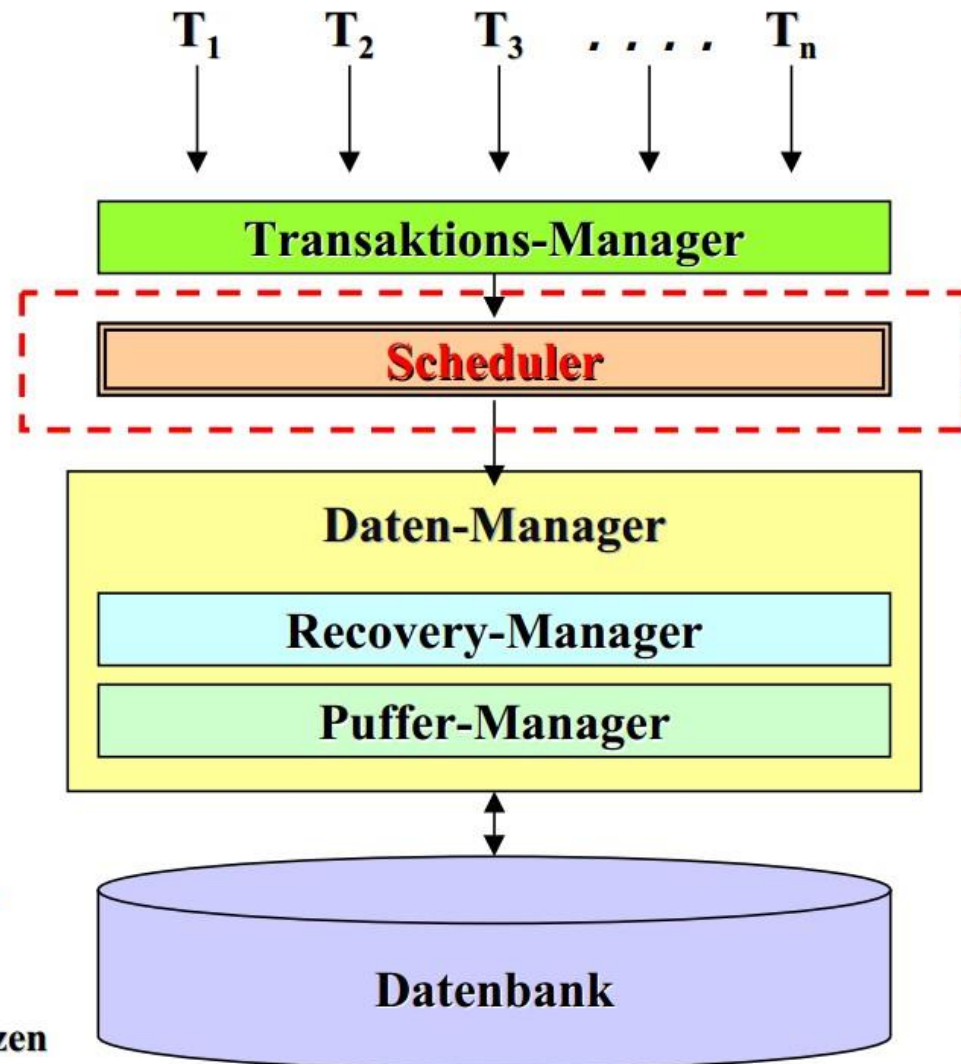


# Mehrbenutzersynchronisation

## Realisierung

- Isolations-Eigenschaft wird durch den **Scheduler** sichergestellt.
- Aufgabe: Koordination des Ablaufs konkurrierender Transaktionen so, dass deren gemeinsame Historie zumindest **serialisierbar** ist !
- **Techniken des Scheduling**:
  - "pessimistisch":
    - **sperrbasierte Synchronisation** (in fast allen kommerziellen DBMS)
    - **zeitstempelbasierte Synchronisation**
  - "optimistisch":
    - Protokollieren und ggf. Rücksetzen

Fokus



(Graphik nach Prof. Kemper)

# Mehrbenutzersynchronisation durch Sperren

## Einleitung

### Grundprinzip der sperrbasierten Synchronisation:

Transaktionen müssen benötigte Daten vor Zugriff durch Setzen einer **Sperre** (engl.: "lock", wörtlich: Schloss) als "Reservierungsvermerk" vor dem Zugriff anderer Transaktionen schützen.

### Sperrmodi:

- **Lesesperre** ("read lock") [symbolisch: **S** von engl. "shared"]
  - Wenn  $T_i$  S-Sperre auf A besitzt, darf  $T_i$  A lesen.
  - Mehrere T. können S-Sperren auf dasselbe A halten.
- **Schreibsperre** ("write lock") [symbolisch: **X** von engl. "exclusive"]
  - Wenn  $T_i$  X-Sperre auf A hat, ist für  $T_i$  auch Schreiben erlaubt.
  - Nur ein  $T_i$  zur Zeit kann eine X-Sperre halten.

### Verträglichkeitsmatrix für Sperranforderungen:

mit NL ("no lock"): ungesperrt,  
✓ : kann gewährt werden,  
– : kann nicht gewährt werden

	NL	S	X
S	✓	✓	–
X	✓	–	–

angefordert

gehalten



# Mehrbenutzersynchronisation durch Sperren

## Allgemeine Grundannahmen

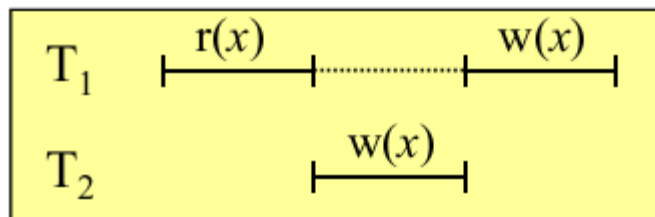
- Sperre ist temporäres Zugriffsprivileg auf einzelnes DB-Objekt
- Anforderung = LOCK, Freigabe = UNLOCK, beides atomare Operationen
- Granularität:
  - komplette DB
  - Schema
  - Tabelle
  - Index
  - Datensatz
  - Datenfeld
  - ...

# Mehrbenutzersynchronisation durch Sperren

## Allgemeine Grundannahmen – legale Schedules

- Jedes Objekt, das von einer TA benutzt werden soll, muss vorher gesperrt werden
- Eine TA fordert eine Sperre, die sie schon besitzt, nicht erneut an
- Am Ende der TA werden alle Sperren freigegeben
- eine TA muss die Sperren anderer TA auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Queue eingereiht – bis die Sperre gewährt werden kann

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



?

**Sind „legale“  
Schedules  
serialisierbar?**



# Mehrbenutzersynchronisation durch Sperren

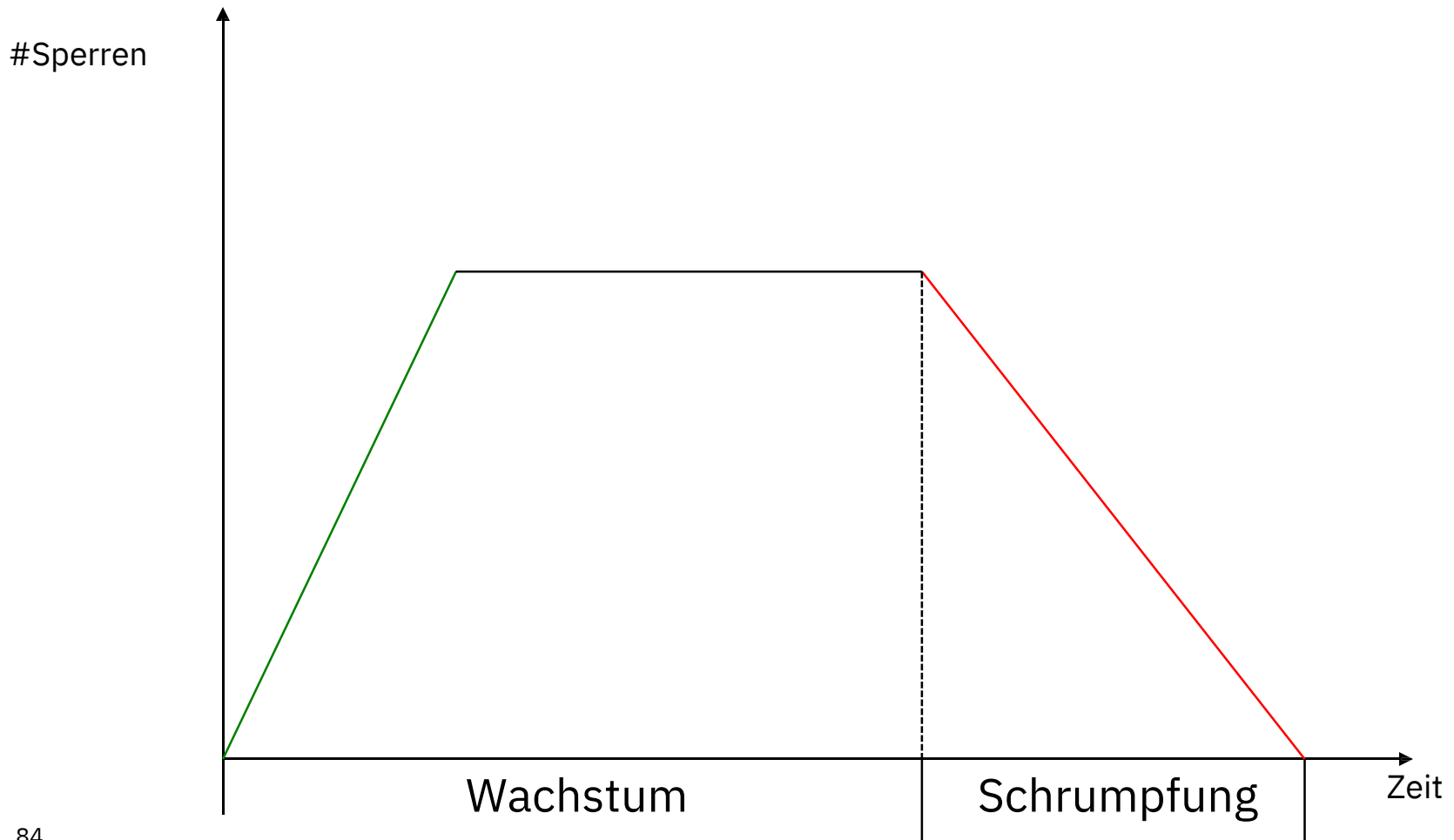
## 2 Phasen-Sperrprotokoll (2PL)

- Allgemein ist eine (in Bezug auf Sperren) legale Historie noch nicht serialisierbar, da Sperren jederzeit beantragt und freigegeben werden können
- Idee ist hier, dass eine „Verzahnung“ von Transaktionen unterbunden wird
- Jede TA durchläuft zwei Phasen:
  - Eine Wachstumsphase: Sperren anfordern, aber keine freigeben
  - eine Schrumpfphase: bisher erworbene Sperren freigeben, keine anfordern
- Dadurch ist eine Serialisierbarkeit garantiert



# Mehrbenutzersynchronisation durch Sperren

## 2-Phasen-Sperrprotokoll (2PL) - Grafik





# Mehrbenutzersynchronisation durch Sperren

## 2-Phasen-Sperrprotokoll (2PL) - Beispiel

- T1 modifiziert nacheinander die Datenobjekte A und B (z.B. eine Überweisung)
- T2 liest nacheinander dieselben Datenobjekte A und B (Z.B. zur Aufsummierung der beiden Kontostände)





# Mehrbenutzersynchronisation durch Sperren

## 2-Phasen-Sperrprotokoll (2PL) - Beispiel

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



# Mehrbenutzersynchronisation durch Sperren

## 2-Phasen-Sperrprotokoll (2PL) – Beispiel 2

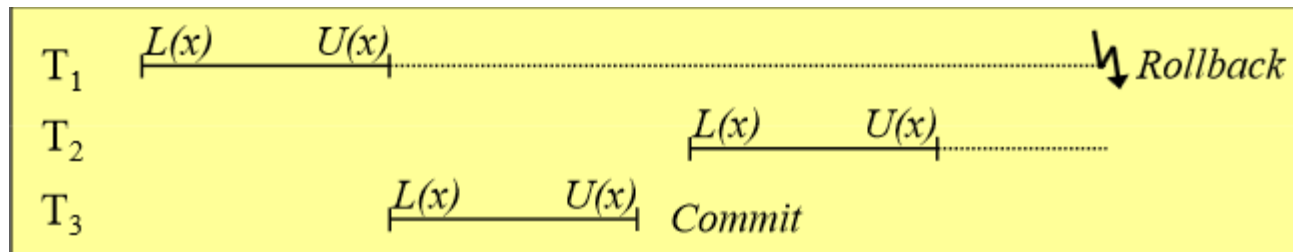
Schritt	T <sub>1</sub>	T <sub>2</sub>	Bemerkung
1.	<b>BOT</b>		
2.	<b>lockX(A)</b>		
3.	read(A)		
4.	write(A)		
5.		<b>BOT</b>	
6.		<b>lockS(A)</b>	T <sub>2</sub> muss warten
7.	<b>lockX(B)</b>		
8.	read(B)		
9.	<b>unlockX(A)</b>		T <sub>2</sub> wecken
10.		read(A)	
11.		<b>lockX(B)</b>	T <sub>2</sub> muss warten
12.	write(B)		
13.	<b>unlockX(B)</b>		T <sub>2</sub> wecken
14.		write(B)	
15.	<b>abort</b>		
16.		<b>unlockS(A)</b>	
17.		<b>unlockX(B)</b>	
18.		<b>commit</b>	



# Mehrbenutzersynchronisation durch Sperren

## 2-Phasen-Sperrprotokoll (2PL) - Beispiel

- Beispiel zeigte einen legalen Schedule nach dem 2PL
  - Wegen 2PL serialisierbar ☺
  - Aber wir hatten kaskadierendes zurücksetzen ☹
  - Noch schlimmer folgendes Beispiel mit einer serialisierbaren Historie, die wir aber nicht zurücksetzen können und somit gegen die Durability-Regel des ACID- Prinzips verstößen (L=Lock, U=Unlock) ☹☹☹

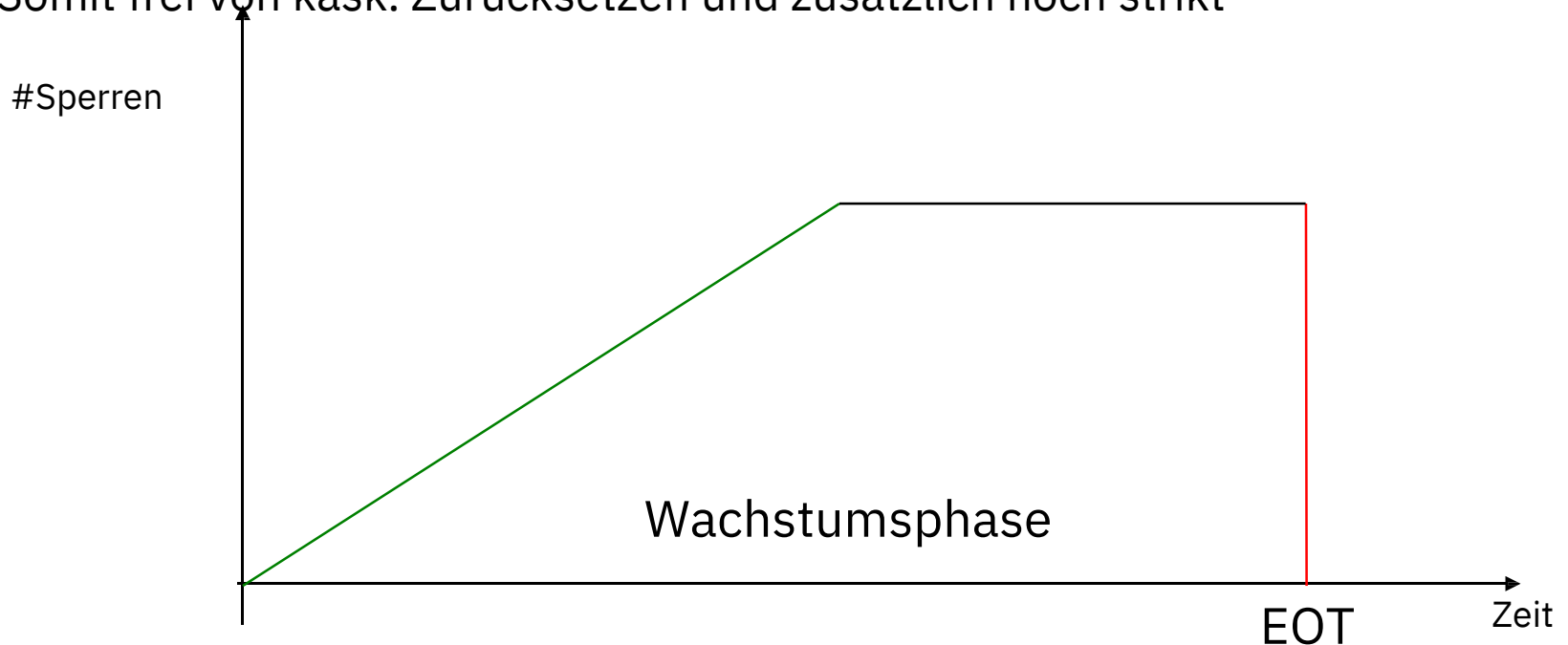




# Mehrbenutzersynchronisation durch Sperren

## strenges 2-Phasen-Sperrprotokoll (S2PL) - Beispiel

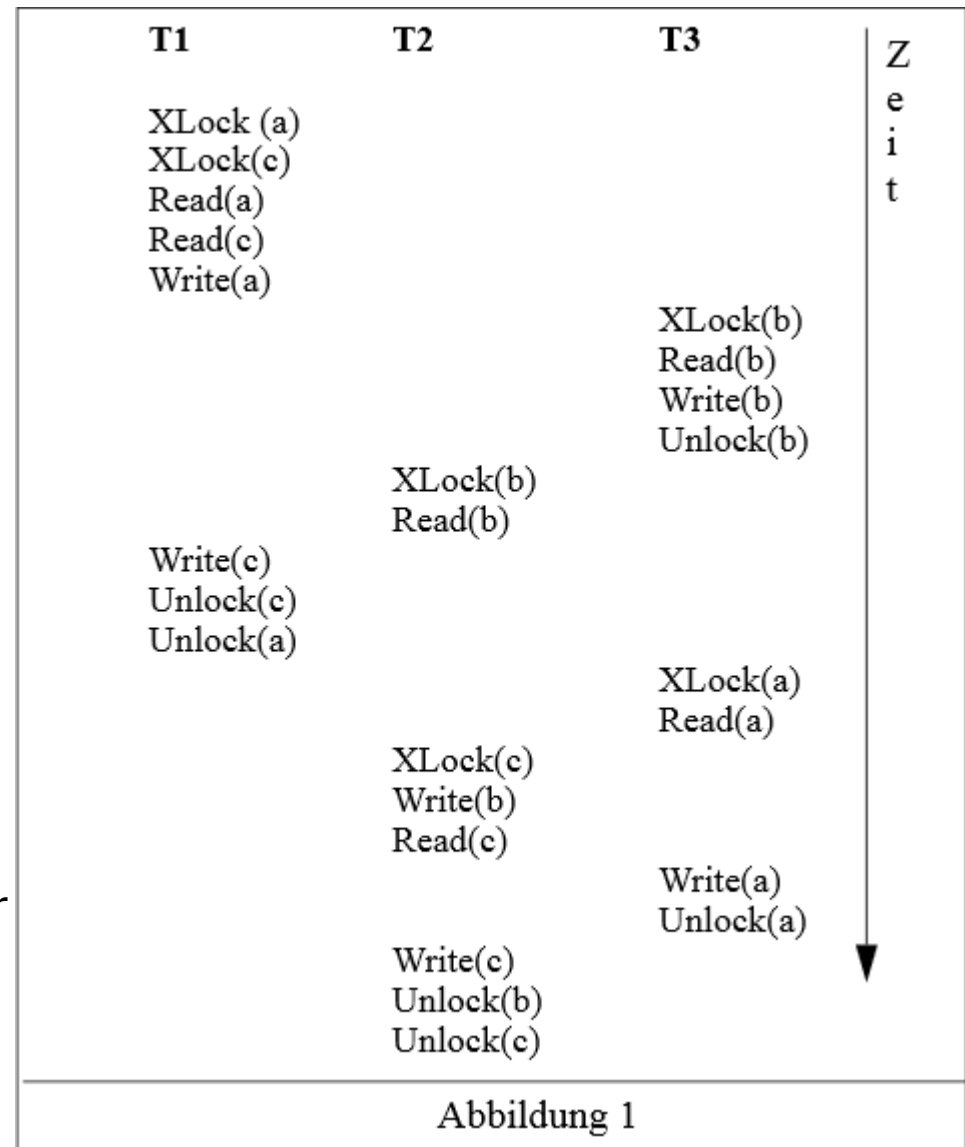
- 2PL schließt kaskadierendes Rücksetzen nicht aus und generiert ggf. nicht-zurücksetzbare Historien. Erweiterung zum strengen (oder striktem) 2PL:
  - alle Sperren werden bis EOT gehalten, dort zusammen atomar freigegeben. Somit frei von kask. Zurücksetzen und zusätzlich noch strikt



# Mehrbenutzersynchronisation durch Sperren

## Übung

- Ist der Schedule legal?
  - Ja, Zugriff nur wenn vorher Sperre beantragt wurde und da nur Xlock sind es auch die richtigen Sperren
- Ist dies ein für das 2PL zulässiger Schedule?
  - Nein, in T3 keine Phasen
- Ist dies ein für das S2PL zulässiger Schedule?
  - Nein, da schon nicht 2PL





# Mehrbenutzersynchronisation durch Sperren

## Deadlocks / Verklemmungen

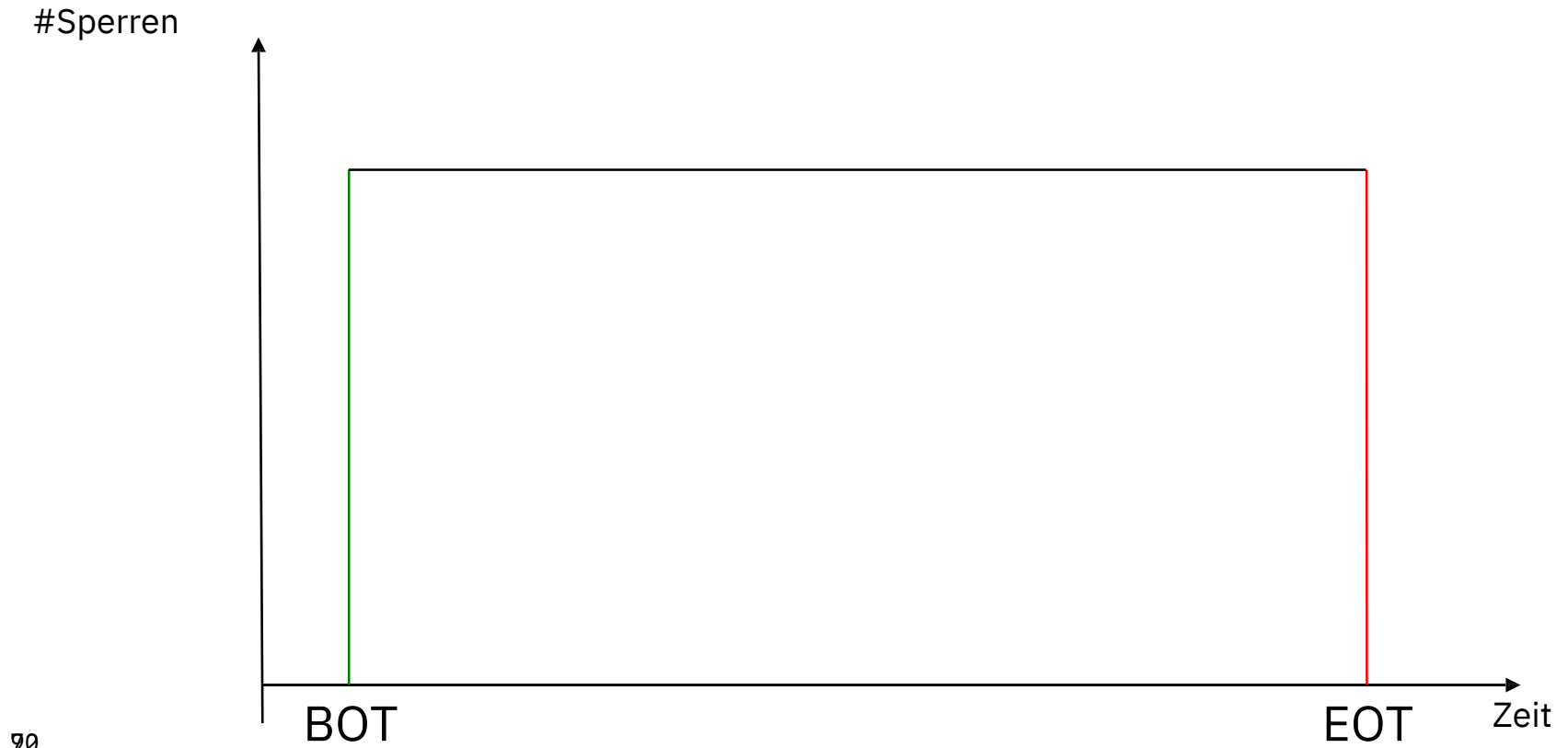
Schritt	$T_1$	$T_2$	Bemerkung
1.	<b>BOT</b>		
2.	<b>lockX(A)</b>		
3.		<b>BOT</b>	
4.		<b>lockS(B)</b>	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	<b>lockX(B)</b>		$T_1$ muss warten auf $T_2$
9.		<b>lockS(A)</b>	$T_2$ muss warten auf $T_1$
10.	...	...	$\Rightarrow$ Deadlock



# Mehrbenutzersynchronisation durch Sperren

## Deadlock-Vermeidung

- S2PL mit „Preclaiming“

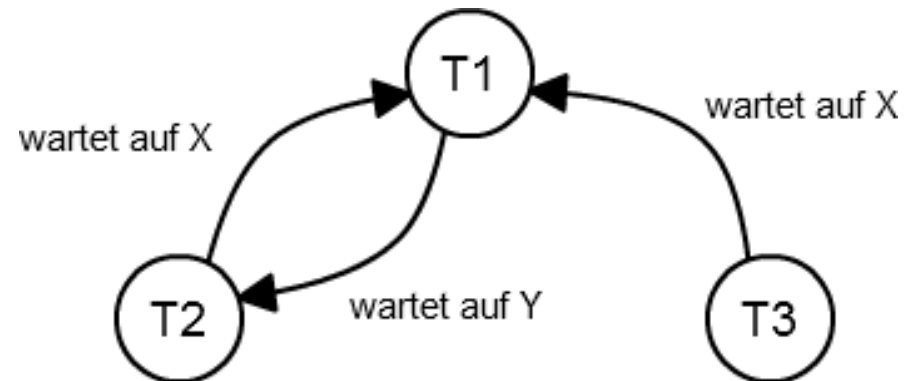


# Mehrbenutzersynchronisation durch Sperren

## Deadlock-Erkennung

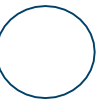
- Wartegraphen

T1	T2	T3
xlock(X)		
	xlock(Y)	
		xlock(X)
xlock(Y)		
	xlock(X)	



- Zyklenerkennung notwendig
  - Mäßig aufwendig...
  - Beim Beispiel oben: Da die meisten TA auf T1 warten, wäre ein Abbruch von T1 von hoher Effektivität





# Mehrbenutzersynchronisation durch Sperren

## Deadlock-Handling

- 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
  - $T_1$  will Sperre erwerben, die von  $T_2$  gehalten wird:
    - wound-wait Strategie
      - 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
      - Wenn  $T_1$  älter als  $T_2$  ist, wartet  $T_1$  auf die Freigabe der Sperre.
      - Sonst wird  $T_1$  abgebrochen und zurückgesetzt.



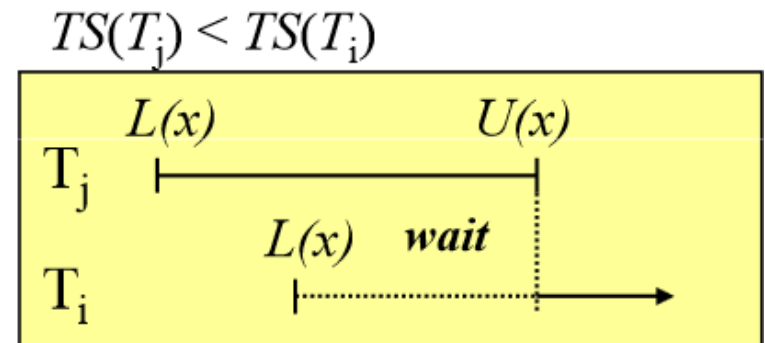
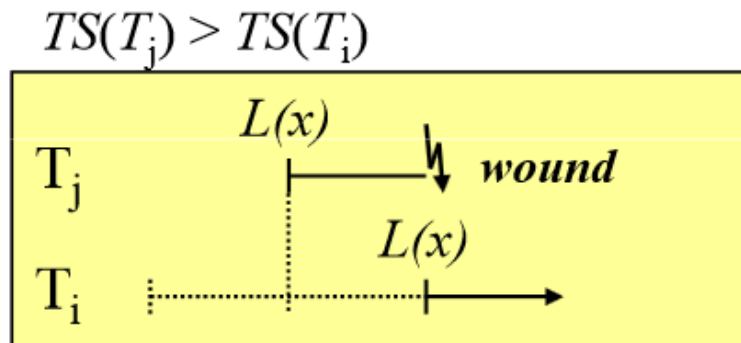
# Mehrbenutzersynchronisation durch Sperren

## Deadlock-Handling: wound-wait

### wound-wait:

$T_i$  fordert Sperre  $L(x)$  an.

- Jüngere TA  $T_j$ , d.h.  $TS(T_j) > TS(T_i)$ , hält bereits Sperre auf  $x$ :  
=>  $T_i$  läuft weiter, jüngere TA  $T_j$  wird zurückgesetzt (**wound**)
- Ältere TA  $T_j$ , d.h.  $TS(T_j) < TS(T_i)$ , hält bereits Sperre auf  $x$ :  
=>  $T_i$  wartet auf Freigabe der Sperre durch ältere TA  $T_j$  (**wait**)



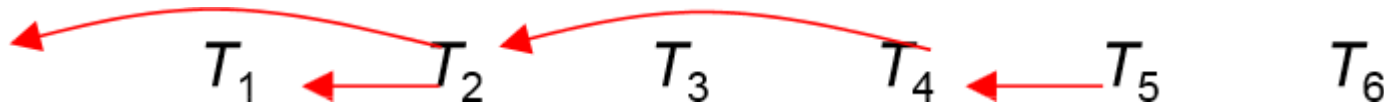
→ ältere TAs „bahnen“ sich ihren Weg durch das System



# Mehrbenutzersynchronisation durch Sperren

## Deadlock-Handling: wound-wait

- Für die Zeitstempel / Timestamps (TS) ergibt sich:
  - $TS(T_1) < TS(T_2) < \dots < TS(T_n)$
- Jüngere TA warten immer auf ältere, nie anders herum. Wartegraph also z. B.:

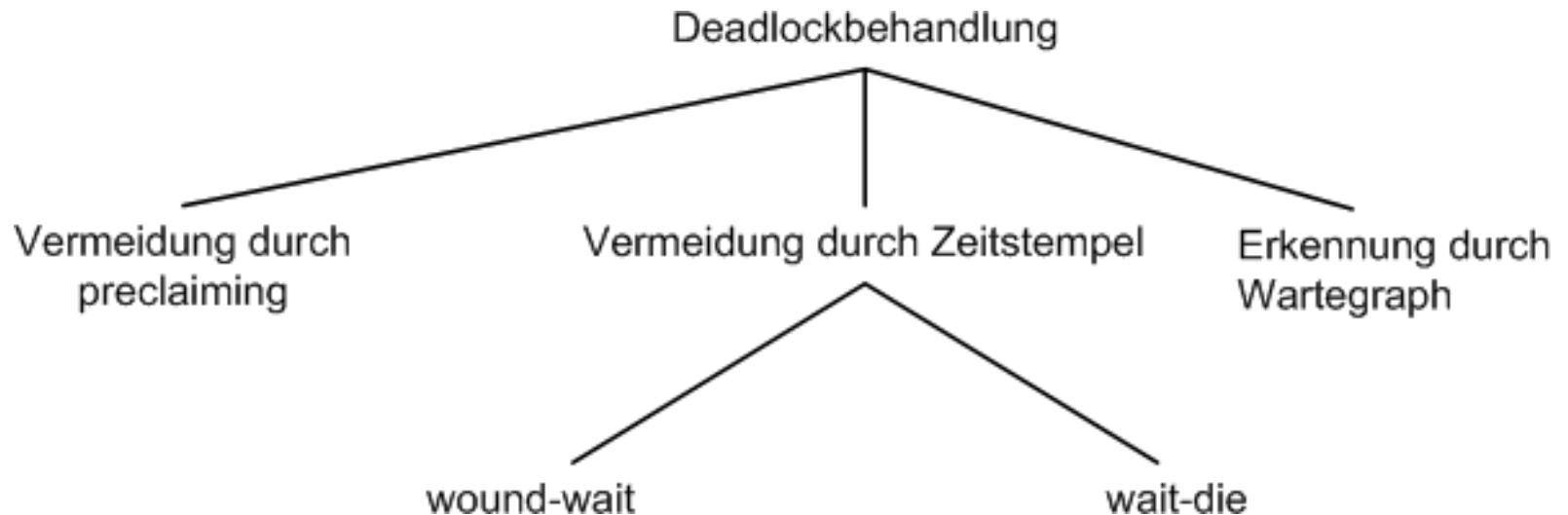


- Es ist erkennbar, dass Pfeile nie von links nach rechts gehen können → somit kann es keine Zyklen geben und der Schedule ist garantiert deadlockfrei
- Serialisierbarkeit trotzdem gegeben, da Sperrprotokolle (S2PL) eingehalten werden
- Bei wait-die ist alles anders herum



# Mehrbenutzersynchronisation durch Sperren

## Deadlock-Behandlung





# Mehrbenutzersynchronisation durch Sperren

## Deadlock-Handling - Zusammenfassung

- Preclaiming in der Praxis sehr schwierig bis unmöglich!
- Zeitstempelverfahren setzen TA ggf. viel häufiger zurück als notwendig
- Wenn man obige Verfahren nicht nutzt, können Deadlocks ggf. entstehen. Praxis?
  - Time-Out: Wenn eine TA x Minuten ununterbrochen wartet, abbrechen und neu starten. Hier muss x sinnvoll gewählt werden!
  - Wartegraphen erkennen nur Deadlocks, welche dann nach einer Regel beendet werden müssen. Mögliche Strategien:
    - Wenig Aufwand: Am besten jüngste TA
    - Maximierung der Ressourcen: die TA mit den meisten Sperren
    - Erfolgversprechend: die TA, die an den meisten Zyklen beteiligt ist
  - Kein Aushungern: In der Vergangenheit zurückgesetzte TA nicht nochmal zurücksetzen
- 98 – Großes Problem: Live-Locks bei ungerechter Rücksetzlogik!



# Mehrbenutzersynchronisation durch Sperren

## Phantom Problem

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



# Mehrbenutzersynchronisation durch Sperren

## Phantom Problem

- Selbst das 2PL mit Preclaiming kann Phantome i. A. nicht verhindern
- Neben den Daten selbst muss auch der Zugriffsweg zu den Daten gesperrt werden (Datensatz -> Range -> Table -> Schema)
- In modernen DBMS nicht nur 3 Lock-Modi sondern dutzende vorhanden ...

# Mehrbenutzersynchronisation

## Isolation Levels in SQL



**Machen Sperren immer  
Sinn?**





# Mehrbenutzersynchronisation

## Isolation Levels in SQL

- Schon recht früh in SQL2 (SQL-92) spezifiziert
- Manchmal kann es zur Vermeidung von Wartezeit sinnvoll sein, gewisse Anomalien zu akzeptieren
- Isolation Levels umfassen 4 Levels, die festlegen, wie lange Objekte in der Datenbank gesperrt werden
- In Oracle z.B.:  
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
- In Db2, Derby etc. kann der gewünschte Level im SELECT festgelegt werden:  
SELECT ... FROM tabelle WITH UR / CS / RS / RR
- Zusätzlich kann festgelegt werden, ob ein Cursor / Resultset nur zum Lesen (FOR READ ONLY) oder auch zum Ändern (FOR UPDATE) bestimmt ist



# Mehrbenutzersynchronisation

## Isolation Levels: Derby-Syntax zum Setzen

```
SET [ CURRENT ] ISOLATION [ = ]  
{ UR | DIRTY READ | READ UNCOMMITTED  
| CS | READ COMMITTED | CURSOR STABILITY  
| RS  
| RR | REPEATABLE READ | SERIALIZABLE  
| RESET  
}
```

### *query*

```
[ ORDER BY clause ]  
[ result offset clause ]  
[ fetch first clause ]  
[ FOR { READ ONLY | FETCH ONLY  
      | UPDATE [ OF simpleColName [ , simpleColName ]* ] }  
[ WITH { RR | RS | CS | UR } ]
```



# Mehrbenutzersynchronisation

**Achtung Begriffsverwirrung!**

JDBC isolation level	Db2 / Derby isolation level	
TRANSACTION_READ_UNCOMMITTED	Uncommitted Read	UR
TRANSACTION_READ_COMMITTED	Cursor Stability	CS
TRANSACTION_REPEATABLE_READ	Read Stability	RS
TRANSACTION_SERIALIZABLE	Repeatable read	RR



# Isolation Levels in SQL

## read uncommitted

- schwächste Konsistenzstufe: Zugriff auf nicht geschriebene Daten
  - nur für read only Transaktionen
- Für statistische Transaktionen: „ungefährer Überblick“
- keine Sperren notwendig, effizient ausführbar, keine Blockierung anderer TA
- TA hat Zugriff auf Daten die noch nicht per commit finalisiert wurden
- dirty read, non repeatable read und Phantome möglich



**Wann ist das sinnvoll?**

T <sub>1</sub>	T <sub>2</sub>
	read(A)
	...
	write(A)
read(A)	
...	
	<b>rollback</b>



# Isolation Levels in SQL

## read committed

- Zugriff auf festgeschriebene Daten
- Für viele Anwendungen ausreichend
  - Da SQL nicht prozedural ist, gilt eine „große“ Abfrage als atomare Einheit
  - Zwischenspeichern von Werten in Variablen ist bei read committed jedoch gefährlich, da:
- non repeatable read und Phantome möglich

T <sub>1</sub>	T <sub>2</sub>
read(A)	write(A) write(B) <b>commit</b>
read(B) read(A)	
...	



# Isolation Levels in SQL

## Repeatable read

- Das 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 nun ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.
- Phantome möglich



# Isolation Levels in SQL

## Serializable

- Garantiert Serialisierbarkeit
  - Prädikatssperren z. B. für Range-Angaben im WHERE-Statement
  - Verhindert alle vorher bekannten Anomalien
  - Häufig nur durch Sperren der gesamten Tabelle realisierbar



# Isolation Levels in SQL

## Übersicht

- Isolationslevel sind die Definition dafür, wie das DBMS im Hintergrund notwendige Sperren managed. Je nach „Art“ der Sperrenverwaltung können unterschiedliche Ergebnisse erzielt werden:

	dirty read	non repeatable read	Phantome
read uncommitted	ja	ja	ja
read committed	nein	ja	ja
repeatable read	nein	nein	ja
serializable	nein	nein	nein

- Das **lost-update-Problem** kann immer dann auftreten, wenn wir einen non repeatable read haben könnten → heißt: sowohl bei read uncommitted als auch bei read committed kann es vorkommen, dass bereits vorgenommene Änderungen an einem Datensatz verloren gehen könnten, siehe z. B. [http://de.wikipedia.org/wiki/Verlorenes\\_Update](http://de.wikipedia.org/wiki/Verlorenes_Update)





# Übungen

## Fragen (falls wir es nicht mehr im Unterricht schaffen: HA!)

- Wir haben gelernt, dass alle 2PL-konformen Schedules serialisierbar sind.  
Zeigen Sie, dass es serialisierbare Schedules gibt, die von einem auf dem 2PL basierenden Scheduler nicht generiert worden sein können
- Existiert zu folgendem Schedule ein konfliktäquivalenter serieller Schedule?  $r1(a), r2(b), r3(a), w3(a), r1(b), w1(b), r2(c), r1(c), r2(d), r1(d), w1(d), w3(c)$
- Reicht es beim S2PL aus, alle Schreibsperrern bis zum Ende zu halten, Lesesperrern aber ggf. schon früher freizugeben?



# Übungen

## Lösungen (falls wir es nicht mehr im Unterricht schaffen: HA!)

- Wir haben gelernt, dass alle 2PL-konformen Historien serialisierbar sind. Zeigen Sie, dass es serialisierbare Historien gibt, die von einem auf dem 2PL basierenden Scheduler nicht generiert worden sein können.

Beispiel:  $H = (w1(a), r2(a), r1(a))$

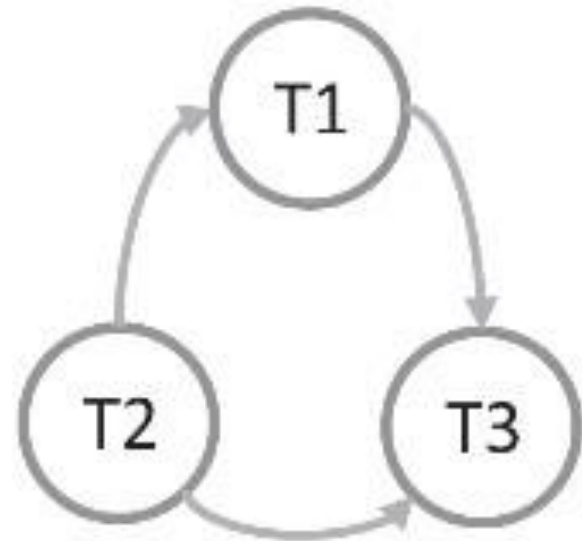
TA1 sperrt A um es zu schreiben. Durch exklusive Sperre auf A muss TA2 warten, bis Sperre aufgehoben wird von TA1. Da TA1 aber später noch  $r1$  liest, kann es die Sperre noch nicht freigeben. Somit kann  $r2(a)$  unmöglich zwischen den beiden anderen Operationen ausgeführt werden, da es keine Sperre beantragen kann. Somit ist obige Historie zwar konfliktäquivalent zur seriellen Abfolge  $TA1 \rightarrow TA2$ , kann aber so nicht von einem 2PL-Scheduler generiert worden sein. (es ist aber durchaus möglich, dass die Befehle beim Transaktionsmanager in dieser Reihenfolge angekommen sind)

# Übungen

## Lösungen (falls wir es nicht mehr im Unterricht schaffen: HA!)

- Existiert zu folgendem Schedule ein konfliktäquivalenter serieller Schedule?  $r_1(a), r_2(b), r_3(a), w_3(a), r_1(b), w_1(b), r_2(c), r_1(c), r_2(d), r_1(d), w_1(d), w_3(c)$

$T_2 \rightarrow T_1 \rightarrow T_3$





# Übungen

## Lösungen (falls wir es nicht mehr im Unterricht schaffen: HA!)

- Reicht es beim S2PL aus, alle Schreibsperrern bis zum Ende zu halten, Lesesperren aber ggf. schon früher freizugeben?
  - „Erweiterung“ zum 2PL ist die Vermeidung von kaskadierendem Zurücksetzen (und noch Gewährleistung strikter Historien)
  - Wann haben wir das? Wenn eine zweite TA auf geänderte Werte der schreibenden ersten TA zugreift. Lesesperren erlauben kein Schreiben von Daten, also kann durch Lesesperren an sich überhaupt kein kaskadierendes Zurücksetzen ausgehen.



# Hausaufgaben Mehrbenutzersynchronisation

## bis zur nächsten Vorlesung

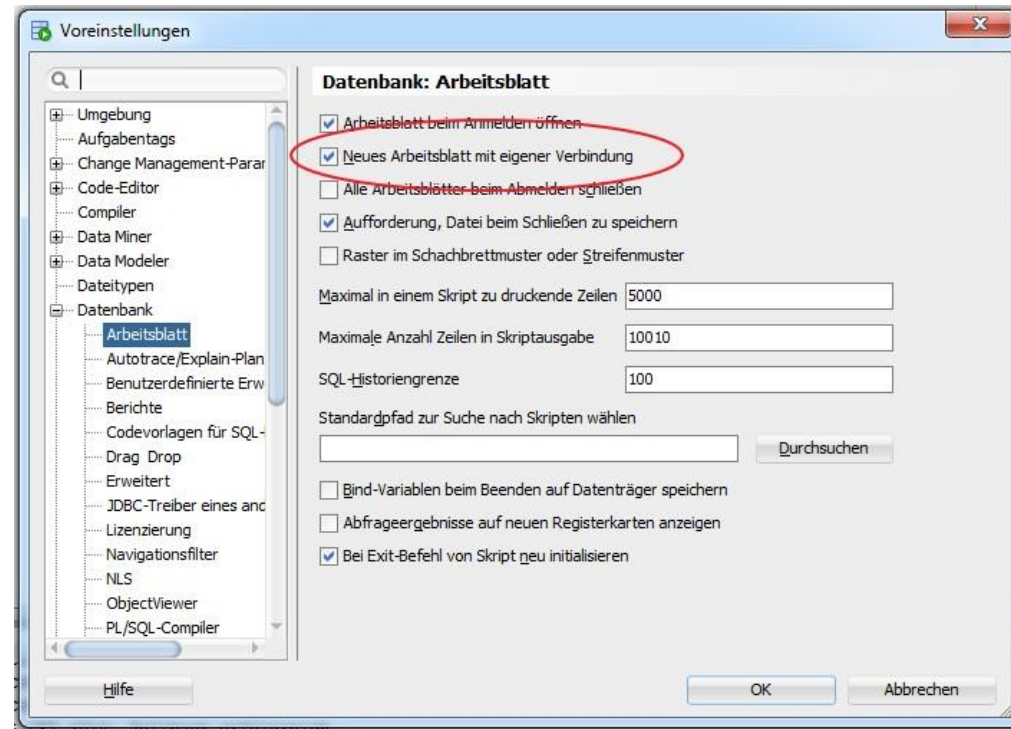
@

Finden Sie mithilfe von parallelen Transaktionen heraus, welches Transaktionsisolutionslevel Oracle standardmäßig eingestellt hat. Achten Sie darauf, das Häkchen (siehe rechts) in den Einstellungen zu setzen bevor Sie mit Transaktionen arbeiten!

@

Generieren Sie einmal sowohl mit dem Transaktionsisolutionslevel „Serializable“ als auch „read committed“ zwischen 2 Transaktionen einen nichtserialisierbaren Schedule (ohne Range-Angaben im WHERE-Statement) in Form eines „lost updates“.

Wie reagiert Oracle?





# Hausaufgaben Mehrbenutzersynchronisation

## bis zur nächsten Vorlesung



**Installieren Sie in der VM ein Tool, welches es Ihnen erlaubt, Java-Anwendungen zu entwickeln**