Einsatz und Realisierung von Datenbanksystemen

ERDB Übungsleitung

Alice Rey, Maximilian Reif, Tobias Götz

i3erdb@in.tum.de

Folien erstellt von Maximilian Bandle & Alexander Beischl

Organisatorisches
Disclaimer

Die Folien werden von der Übungsleitung allen Tutoren zur Verfügung gestellt.

Sollte es Unstimmigkeiten zu den Vorlesungsfolien von Prof. Kemper geben, so sind die Folien aus der Vorlesung ausschlaggebend.

Falls Ihr einen Fehler oder eine Unstimmigkeit findet, schreibt an i3erdb@in.tum.de mit Angabe der Foliennummer.

Mehrbenutzersynchronisation
Formale Definition einer Transaktion

Operationen einer Transaktion TA T_i

- BOT_i Beginn der Transaktion (Begin Of Transaction)
- $\mathbf{r_i(A)}$ Lesen (Read) von Datenobjekt A
- $\mathbf{w_i(A)}$ Schreiben (Write) von Datenobjekt A
- $\mathbf{a_i}$ Abbruch (Abort) der Transaktion
- $\mathbf{c_i}$ Festschreiben (Commit) der Transaktion

Mehrbenutzersynchronisation
Konfliktoperationen

In Konflikt stehende Operationen dürfen nicht parallel ausgeführt werden

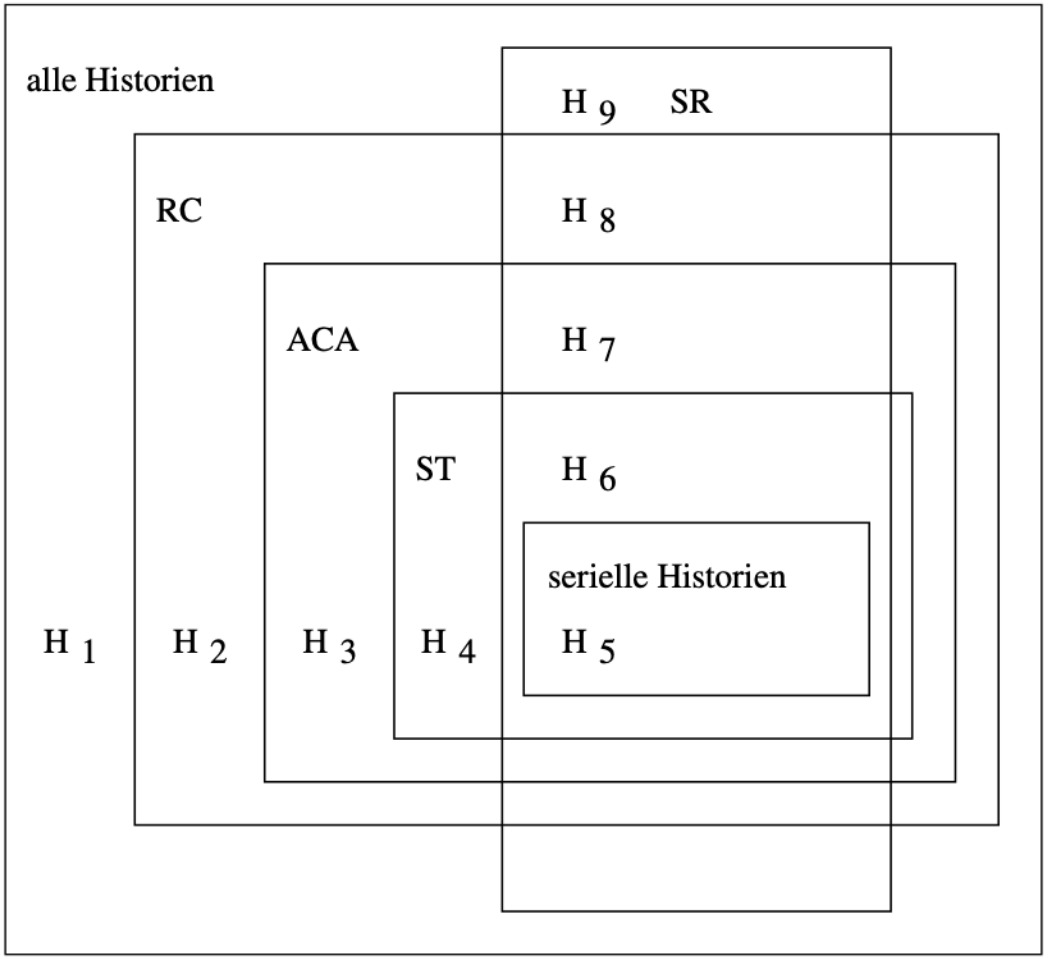
Zwei Operationen stehen in Konflikt, wenn beide auf dem selben Datenobjekt arbeiten wollen und mindestens eine Operation schreibt

T_j \ T_i	$r_i[x]$	$w_i[x]$
$r_j[x]$	✔	✘
$w_j[x]$	✘	✘

✔ Kein Konflikt
✘ Konflikt

Mehrbenutzersynchronisation
Klassifikation von Historien

SR: serialisierbar
RC: rücksetzbar
ACA: vermeidet kaska-
dierendes Rücksetzen
ST: strikt
ST&SR: seriell



Mehrbenutzersynchronisation
Klassifikation von Historien (Serialisierbar)

Konstruktion eines **Serialisierbarkeitsgraph (SG)**:

1. Jede committete **Transaktion** ist ein **Knoten** im Graph

2. Jede **Konfliktoperationen** K_{ij} ist eine **Kante** im Graph
   ➡ Zeichne einen Pfeil von T_i nach T_j, wenn $\sigma_i < \sigma_j$

➡ Eine Historie ist genau dann **serialisierbar**,
wenn der zugehörige SG **azyklisch** ist!

Mehrbenutzersynchronisation
Klassifikation von Historien (Serialisierbar)

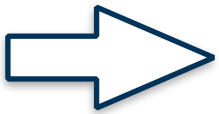
r2[y], r1[y], w2[y], c2, r3[x], w1[x], r3[y], c3, c1

T_1		$r_1[y]$				$w_1[x]$			c1
T_2	$r_2[y]$		$w_2[y]$	c2					
T_3					$r_3[x]$		$r_3[y]$	c3	

Serialisierbar: Serielle Reihenfolge der Ausführung möglich

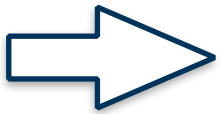
Konfliktoperationen

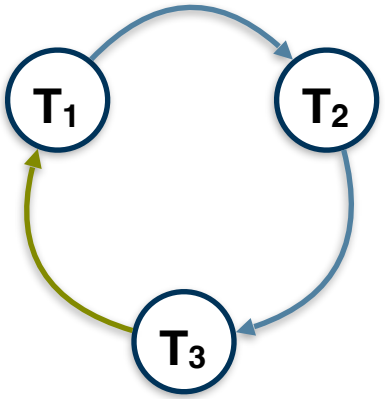
$r_1[y] < w_2[y]$
$w_2[y] < r_3[y]$
$r_3[x] < w_1[x]$



Auswertungsreihenfolge

1 vor 2
2 vor 3
3 vor 1





TAs zyklisch voneinander abhängig => Nicht Serialisierbar

Mehrbenutzersynchronisation
Klassifikation von Historien (Rücksetzbar)

T_1		$r_1[y]$				$w_1[x]$			c1
T_2	$r_2[y]$		$w_2[y]$	c2					
T_3					$r_3[x]$		$r_3[y]$	c3	

RC (**Re**C**o**verable): **Schreiber von Daten muss vor Leser commiten**

Konfliktoperationen

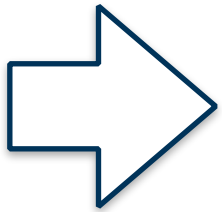
~~$r_1[y] < w_2[y]$~~

$w_2[y] < r_3[y]$

~~$r_3[x] < w_1[x]$~~

Commit-Reihenfolge

$c_2 < c_3 < c_1$



Gewünschte C-Reihenfolge

2 vor 3

Tatsächliche C-Reihenfolge

2 vor 3

Bedingung erfüllt => Rücksetzbar

Mehrbenutzersynchronisation
Klassifikation von Historien (ACA)

T_1		$r_1[y]$				$w_1[x]$			c1
T_2	$r_2[y]$		$w_2[y]$	c2					
T_3					$r_3[x]$		$r_3[y]$	c3	

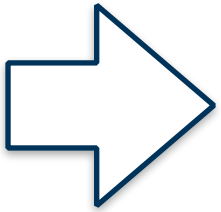
ACA (**A**voiding **C**ascading **A**borts)**:**
Schreiber von Daten muss commiten bevor Daten gelesen werden

Konfliktoperationen

~~$r_1[y] < w_2[y]$~~

$w_2[y] < r_3[y]$

~~$r_3[x] < w_1[x]$~~



geforderte Reihenfolge

$w_2[y] < c_2 < r_3[y]$

Geforderte Reihenfolge wird eingehalten =>
Vermeidet Kaskadierendes Rücksetzen

Mehrbenutzersynchronisation
Klassifikation von Historien (Strikt)

T_1		$r_1[y]$				$w_1[x]$			c1
T_2	$r_2[y]$		$w_2[y]$	c2					
T_3					$r_3[x]$		$r_3[y]$	c3	

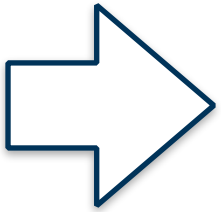
ST (STrict): Schreiber von Daten muss commiten/aborten bevor Daten gelesen oder geschrieben werden

Konfliktoperationen

~~$r_1[y] < w_2[y]$~~

$w_2[y] < r_3[y]$

~~$r_3[x] < w_1[x]$~~



geforderte Reihenfolge

$w_2[y] < c_2 < r_3[y]$

Geforderte Reihenfolge wird eingehalten => Strikt

Mehrbenutzersynchronisation
Aufgabe 1

1. Geben Sie alle Eigenschaften an, die von der Historie erfüllt werden.

$$w_1(x), r_2(y), w_3(y), w_2(x), w_3(z), c_3, w_1(z), c_2, c_1$$

2. Geben Sie alle Eigenschaften an, die von der Historie erfüllt werden.

$$r_1(x), r_1(y), w_2(x), w_3(y), r_3(x), a_1, r_2(x), r_2(y), c_2, c_3$$

3. Gegeben die unvollständige Historie:

$$H = w_1(x), w_1(y), r_2(x), r_2(y)$$

   a) Fügen Sie `commits` in H so ein, dass die Historie RC aber nicht ACA erfüllt:

   b) Fügen Sie `commits` in das ursprüngliche H so ein, dass die Historie ACA erfüllt.

Mehrbenutzersynchronisation
Tool zum Üben

transactions.db.in.tum.de

Mehrbenutzersynchronisation
Scheduler

Ordnet eingehende Operationen um serielle & rücksetzbare Historien zu garantieren

Mehrere Möglichkeiten bei Eingang von Operationen

- Sofortige (unverzögerte) Ausführung
- Zurückweisen => Abbruch der Ausführung
- Verzögerung der Ausführung

Zwei Kategorien von Schedulern

- Optimistisch: Möglichst schnelle Ausführung
- Pessimistisch: Möglichst geschickte Ausführung

Mehrbenutzersynchronisation
Optimistische Verfahren

Schnelle Ausführung & anschließendes Aufräumen der Probleme

Snapshot Isolation

- Jede Transaktion arbeitet in einer eigenen Version der Datenbasis
- Beim Commit werden die Änderungen eingefügt
  => Beim Überschneiden des Write-Sets wird abgebrochen

Mehrbenutzersynchronisation
Pessimistische Verfahren - Zeitstempelbasiert

Stoppen der laufenden TA bei Problemen

=> keine Probleme im Anschluss

Zeitstempelbasierte Synchronisation

- Für jedes Datum wird gespeichert wann es zuletzt gelesen oder geschrieben wurde
- Für jede Transaktion wird gespeichert wann sie gestartet wurde

=> Wenn eine TA auf einen neueren Zeitstempel trifft => Abbruch

Mehrbenutzersynchronisation
Pessimistische Verfahren - Sperrbasiert

Überlegte langsamere Ausführung & keine Probleme im Anschluss

Sperrbasierter Scheduler

- Jedes Datenobjekt hat eine Sperre
- Vor dem Zugriff muss die TA eine Sperre für das Objekt anfordern
- Falls das Objekt bereits gesperrt ist muss die TA warten
- Nach Abschluss der Operation wird die Sperre freigegeben

Verträglichkeits-/Kompatibilitätsmatrix

	gehaltene Sperre		
angeford. Sp.	keine	*S*	*X*
S	√	√	–
X	√	–	–

gleichzeitiges Lesen

Shared Lock (Lesesperre)

E**X**clusive Lock (Schreibsperre)

Mehrbenutzersynchronisation
Aufgabe 2

a) Erläutern Sie kurz die zwei Phasen des 2PL-Protokolls.

b) Inwiefern unterscheidet sich das *strenge* 2PL?

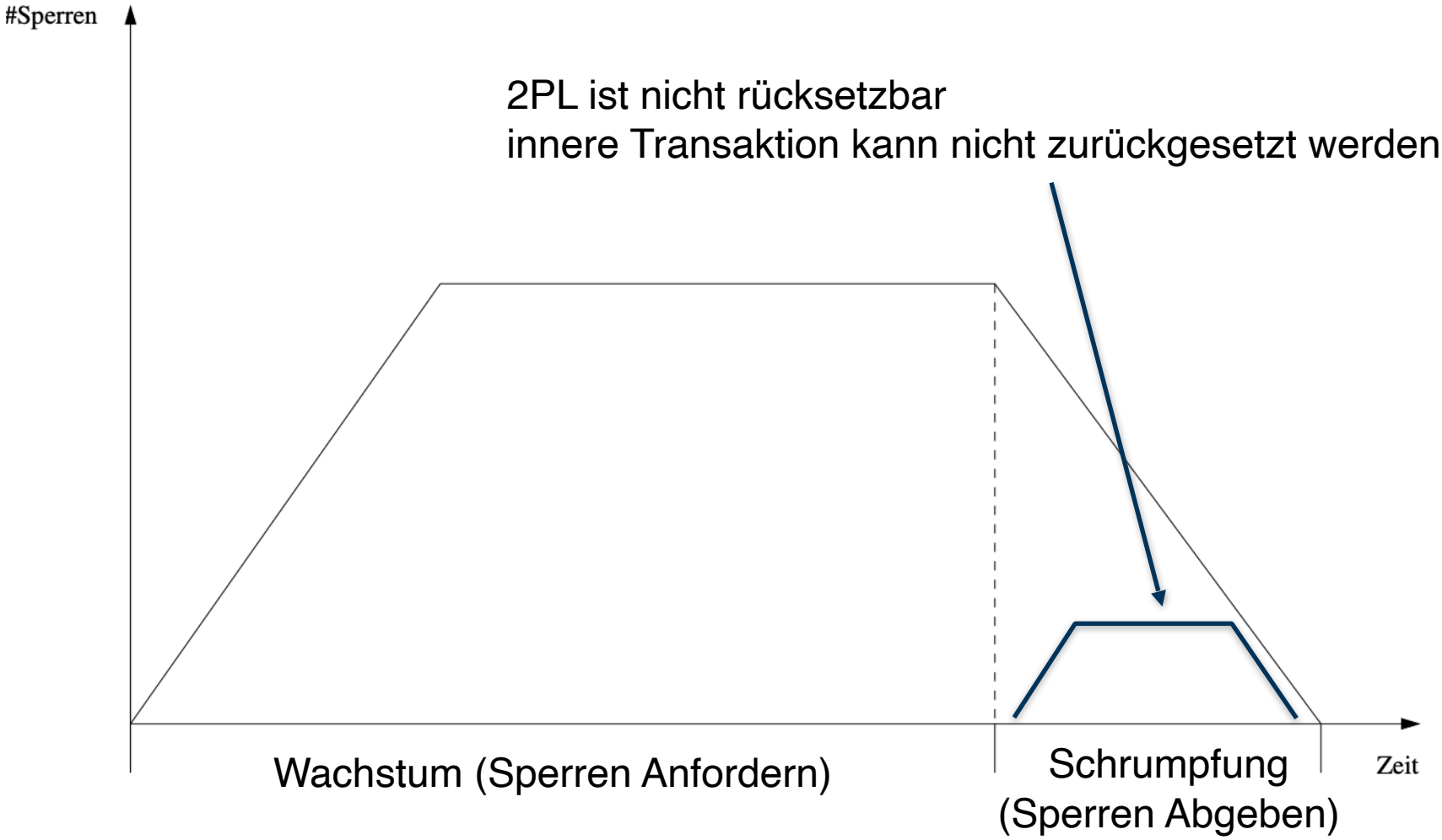
c) Welche Eigenschaften (SR,RC,ACA,ST) haben Historien, welche vom 2PL und vom strengen 2PL zugelassen werden?

d) Wäre es beim strengen 2PL-Protokoll ausreichend, alle Schreibsperren bis zum EOT (Transaktionsende) zu halten, aber Lesesperren schon früher wieder freizugeben?

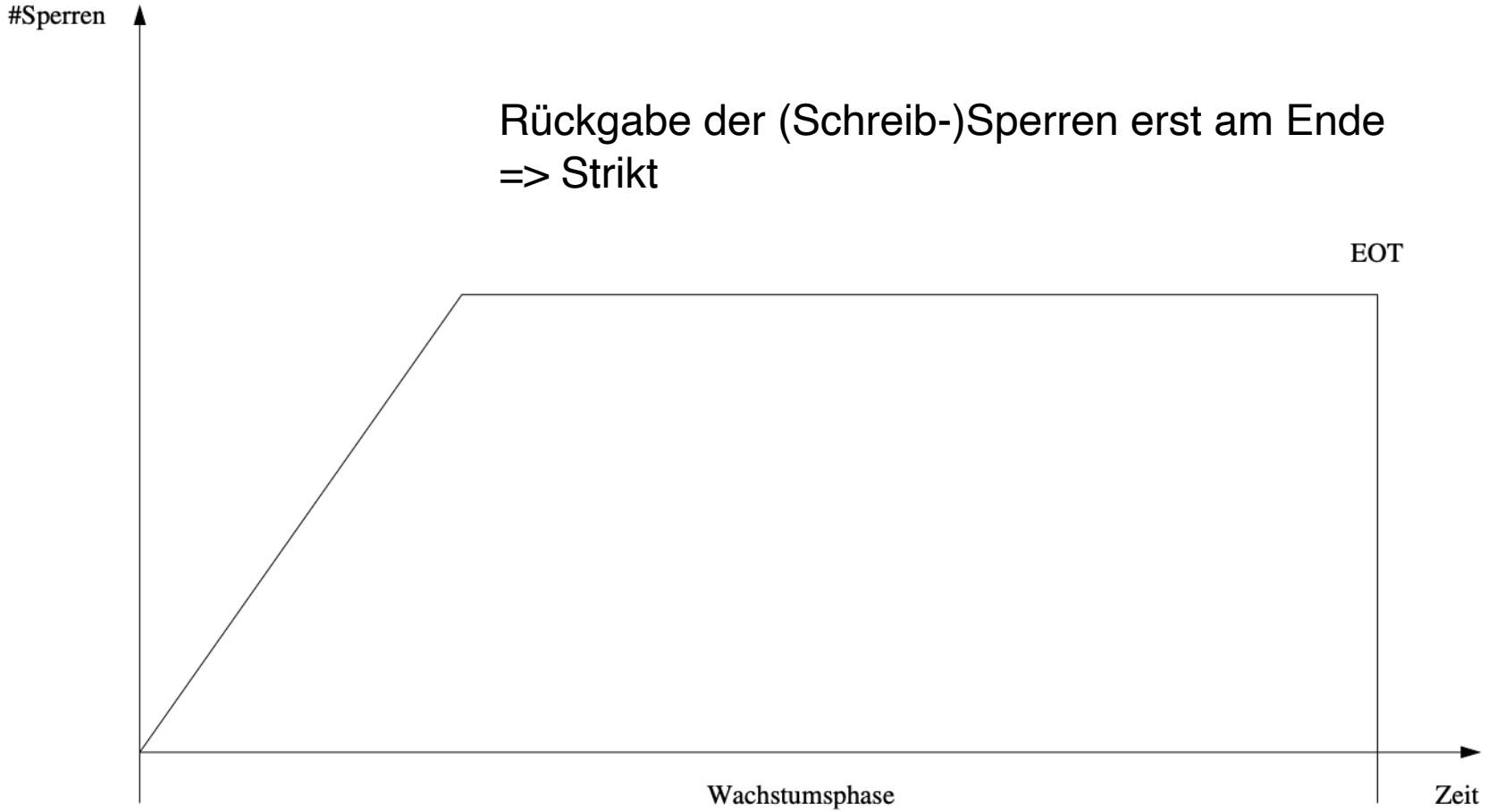
Mehrbenutzersynchronisation
2PL (Two Phase Locking)

- Jedes Objekt das benutzt werden soll, muss vorher gesperrt werden
- Eine TA kann eine Sperre nur einmal anfordern
- Wenn ein Objekt nicht gesperrt werden kann, reiht sich die TA in die Warteschlange ein
- Eine TA darf nach der Freigabe der ersten Sperre keine Sperren anfordern
- Bei Ende der TA müssen alle Sperren zurückgegeben werden

Mehrbenutzersynchronisation
2PL (Two Phase Locking)



Mehrbenutzersynchronisation
Strenges 2PL



Mehrbenutzersynchronisation
Motivation Isolation Levels

Moderne Prozessoren werden kaum schneller, sondern haben mehr Kerne

Serielle Ausführung nicht mehr performant => Parallelisieren der TAs

Folgende Probleme treten bei unzureichender Absicherung auf:

- Lost Update
- Dirty Read
- Non Repeatable Read
- Phantom Problem

Mehrbenutzersynchronisation
Lost Update

T_1	T_2
BOT	
r1(x)	
	BOT
	$r_2(x)$
$w_1(x)$	
	$w_2(x)$
commit	
	commit

Das Ergebnis der Transaktion T_1 ist verloren gegangen!

Mehrbenutzersynchronisation
Dirty Read

Bricht nach Commit von 1 ab

T_1	T_2
BOT	
	BOT
	$r_2(x)$
	$w_2(x)$
$r_1(x)$	
$w_1(y)$	
commit	
	abort

Read vor Commit von T_2

Abort nach Commit von

T_1 liest einen Wert für x der so nicht gültig ist!

Mehrbenutzersynchronisation
Non Repeatable Read

T_1	T_2
BOT	
$r_1(x)$	
	BOT
	$w_2(x)$
	commit
$r_1(x)$	
…	

Verändert Wert während T_1

T_1 liest x zweimal mit verschiedenem Ergebnis!

Mehrbenutzersynchronisation
Phantom Problem

T_1	T_2
BOT	
select count(*) from R;	
	BOT
	insert into R …;
	commit
select count(*) from R;	
…	

Verändert Datenbasis während T_1

T_1 findet ein weiteres Tupel beim Abarbeiten der zweiten Anfrage!

Mehrbenutzersynchronisation
Aufgabe 3

SQL-92 spezifiziert mehrere Konsistenzstufen (*isolation level*) durch welche der Benutzer (bzw. die Anwendung) festlegen kann, wie "stark" eine Transaktion von anderen parallel laufenden Transaktionen isoliert werden soll.

a) Erläutern Sie kurz die Isolation Level. Geben Sie an, welche Nebenläufigkeitsprobleme mit dem jeweiligen Level vermieden werden. Füllen Sie folgende Tabelle aus, die zeigt, welche Probleme durch die jeweiligen Isolation Level verhindert (✓) werden:

		lost update	dirty read	non-repeatable read	phantom problem
isolation level	read un-commited				
	read commited				
	repeatable read				
	serializable				

b) Warum kann zwischen den Konsistenzstufen gewählt werden?

Mehrbenutzersynchronisation
Isolation Levels

		lost update	dirty read	non-repeatable read	phantom problem
isolation level	read un-commited	✓			
	read commited	✓	✓		
	repeatable read	✓	✓	✓	
	serializable	✓	✓	✓	✓

Mehrbenutzersynchronisation
Aufgabe 4

Ein inhärentes Problem der sperrbasierten Synchronisationsmethoden ist das Auftreten von Verklemmungen (Deadlocks). Zur Erkennung von Verklemmungen wurde der Wartegraph eingeführt. Dabei wird eine Kante $T_i \rightarrow T$ eingefügt, wenn T_i auf die Freigabe einer Sperre durch T wartet.

Skizzieren Sie einen Ablauf von Transaktionen, bei dem ein Deadlock auftritt, der einen Zyklus mit einer Länge von mindestens 3 Kanten im Wartegraphen erzeugt.

Mehrbenutzersynchronisation
Aufgabe 5

```
dienstende(arzt_name)
  select count(*) into anzahl_bereit from aerzte where bereit='ja'
  if anzahl_bereit > 1 then
    update aerzte set bereit='nein' where name=arzt_name
```

Die Transaktion soll sicherstellen, dass immer mindestens ein Arzt bereit ist.

Betrachten Sie einen Ablauf, bei dem zwei zur Zeit bereite Ärzte zum gleichen Zeitpunkt entscheiden, ihren Status auf „nein “, d.h. nicht bereit zu ändern:

T_1: execute dienstende('House')
T_2: execute dienstende('Brinkmann')

Gehen Sie beispielsweise davon aus, dass das DBMS versucht, die Transaktion jeweils abwechselnd zeilenweise abzuarbeiten.

Diskutieren Sie:

a) Was kann bei Snapshot Isolation passieren?

b) Warum ist dies bei optimistischer Synchronisation nicht möglich?

c) Wie verhält sich die Zeitstempel-basierte Synchronisation?

d) Wie verhält sich das strenge 2PL?

Name	*Vorname*	*…*	*Bereit*
House	Gregory	…	ja
Green	Mark	…	nein
Brinkmann	Klaus	…	ja

Mehrbenutzersynchronisation
Aufgabe 5a, 5b

a) **Snapshot Isolation**

Ausführungsphase

Führe die Änderungen lokal (im Snapshot) durch

↓

Validierungsphase

Für alle parallelen, commiteten Transaktionen:

Ist WriteSet disjunkt?

$WriteSet(T_i) \cap WriteSet(T_j) = \varnothing$

Ja → **commit**

Nein → **abort**

Jeder Arzt ändert nur sein eigenes Tupel
=> schreiben disjunkt, da für beide die Bedingung erfüllt ist

b) **Optimistische Synchronisation**

Ausführungsphase

Führe die Änderungen lokal (im Snapshot) durch

↓

Validierungsphase

Wie Snapshot Isolation und zusätzlich ReadSet und WriteSet disjunkt

$WriteSet(T_i) \cap ReadSet(T_j) = \varnothing$ und

$ReadSet(T_i) \cap WriteSet(T_j) = \varnothing$

Ja → **commit**

Nein → **abort**

=> Die TA welche die Validierungsphase zuerst erreicht, gewinnt!

Der Rest wird zurückgesetzt

Mehrbenutzersynchronisation
Aufgabe 5b - Optimistische Synchronisation

T_H: ReadSet := Gesamte Datenbasis
WriteSet := Tupel mit House

T_B: ReadSet := Gesamte Datenbasis
WriteSet := Tupel mit Brinkmann

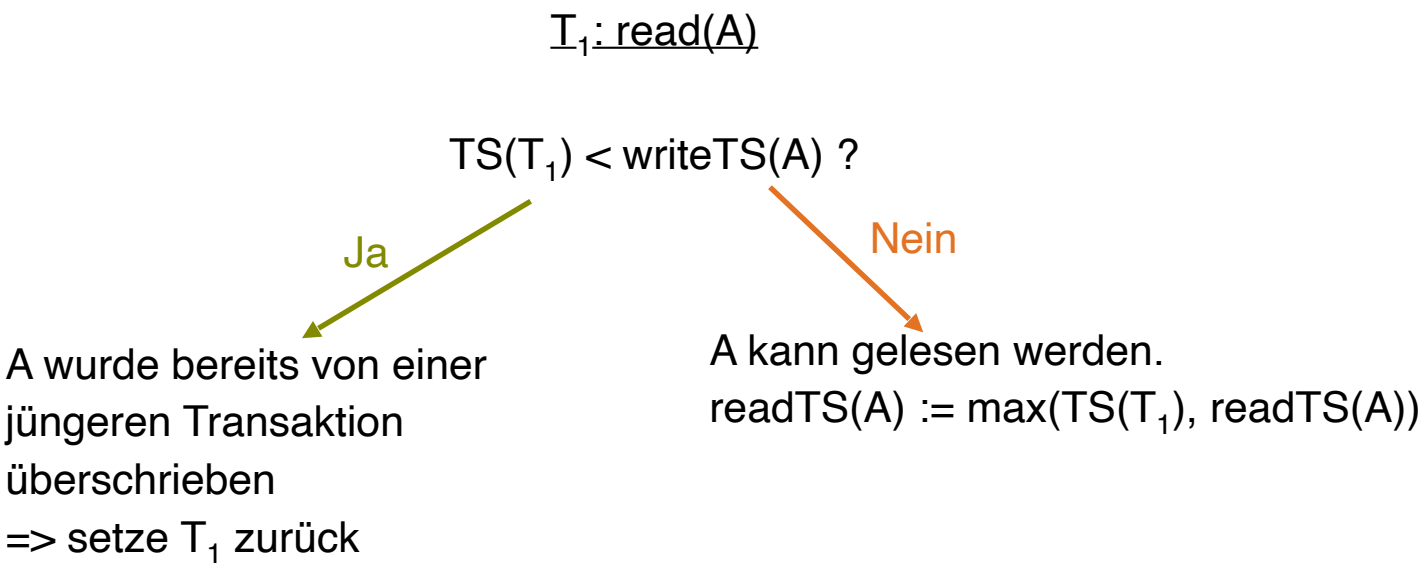
=> **Validierung**: Konflikt, da die Read- & WriteSets nicht disjunkt sind!

$WriteSet(T_H) \cap ReadSet(T_B)$ = Tupel mit House
$WriteSet(T_B) \cap ReadSet(T_H)$ = Tupel mit Brinkmann

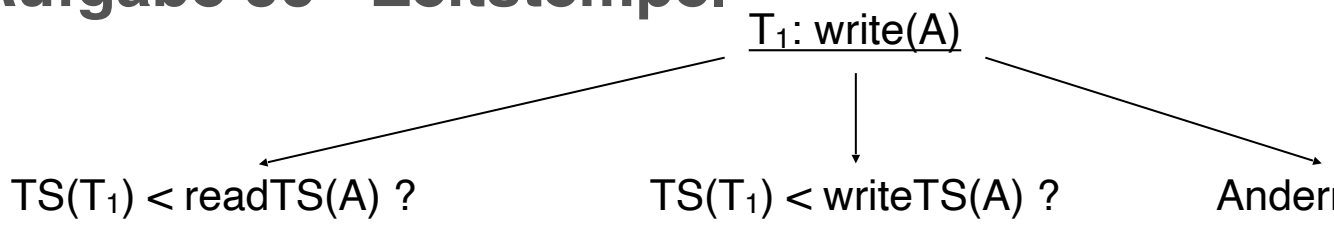
Allerdings: WriteSets sind in der Praxis deutlich kleiner als ReadSets
=> Validierung von *Snapshot Isolation* deutlich günstiger

Mehrbenutzersynchronisation
Aufgabe 5c - Zeitstempel

- Jede TA bekommt zu Beginn (BOT) einen streng monoton aufsteigenden Zeitstempel
- Jedes Datum hat einen Lesestempel (readTS) und einen Schreibstempel (writeTS)
- Wird auf ein Datum zugegriffen, so wird folgendes geprüft:



Mehrbenutzersynchronisation
Aufgabe 5c - Zeitstempel



Der Wert wurde von einer jüngeren TA gelesen, sodass T_1 nicht berücksichtigt würde:
=> setze T_1 zurück

A wurde von einer jüngeren Transaktion geschrieben.
-> TA würde einen jüngeren Wert überschreiben
=> nicht zulässig, setze T_1 zurück

T_1 darf schreiben.
writeTS(A) = TS(T_1)

Angenommen TS(T_H) = 1 und TS(T_B) = 2.

Nachdem beide Ärzte den Status “bereit” abgefragt haben, haben alle Tupel readTS=2.
Anschließend will T_H das Tupel [House,…] ändern.

Da TS(T_H) (=1) < readTS([House,…]) (=2), wird T_H zurückgesetzt und T_B kann nach Hause gehen

Mehrbenutzersynchronisation
Aufgabe 5d - Strenger 2PL

Das Problem tritt nicht auf.

1. Beide erwerben zunächst eine Lesesperre auf alle Bereitschaftsfelder

2. Dann versuchen beide eine Schreibsperre zu bekommen:
   => Deadlock Zykluslänge 2

3. DBMS löst Deadlock: Eine der beiden TA wird abgebrochen

Mehrbenutzersynchronisation
Deadlock-Erkennung und Vermeidung

Timeouts

Transaktionen werden ab einer bestimmten Zeit abgebrochen

=> Deadlocks werden spät erkannt oder korrekte TAs abgebrochen

Wartegraphen

Die zyklische Abhängigkeit der TAs wird überprüft

=> Theoretisch ideale Lösung allerdings teuer zu berechnen

Preclaiming

Alle Sperren werden vor Beginn der TA angefragt

=> Vermindert den Grad der Parallelität drastisch

Deadlockvermeidung durch Zeitstempel

TAs werden je nach Startzeit abgebrochen um Deadlocks zu vermeiden

=> Viele False Positives und Live Lock möglich

Sicherheitsaspekte

Sicherheitsaspekte
k-Anonymität

- Nur Aggregatanfragen erlaubt (SUM, COUNT, AVG…)

- Mindestens k Datensätze müssen aggregiert werden

```
create view HärteDerVorlesung(VorlNr, Härte) as
  select VorlNr, avg(Note)
  from prüfen
  group by VorlNr
  having count(*) > 11;
```

Aufgabe 6

Die AMU (Alexander-Maximilians-Universität) hat eine Datenbank mit den Durchschnittsnoten aller Studenten mit Name.

Schema: {[Name | Durchschnittsnote]}

Der einzige Schutzmechanismus dieser Datenbank ist, dass immer mindestens 3 Tupel aggregiert werden. Als Ausgabe sind nur `COUNT` und `AVG` zulässig.

1. Beschreibe eine Methode, um herauszufinden, was die Note des schlechtesten Studenten der AMU ist
2. Max will Alex' Durchschnittsnote herausfinden. Dazu stellt er folgende Anfragen mit Ergebnis:

   ```
   SELECT AVG(Durchschnittsnote), COUNT(*) FROM Noten => (2,5 ; 10.000)
   SELECT AVG(Durchschnittsnote), COUNT(*) FROM Noten WHERE Name != 'Alex'
   => (2,5001 ; 9.999)
   ```

   Kann er aus den Ergebnissen Alex' Note berechnen? Wenn ja, wie, wenn nein, wieso nicht?

Fragen?