



Kapitel 08 – Transaktionsverwaltung

Vorlesung Datenbanken

Prof. Dr. Kai Höfig



Kapitel 08: Transaktionen

In diesem Kapitel wollen wir folgende Fragen betrachten

- Transaktionen:
 - Was passiert, wenn mehrere Nutzer gleichzeitig auf eine Datenbank zugreifen wollen?
 - Was ist eine Transaktion, und wofür brauche ich sie?
 - Was sind die ACID Kriterien?
 - Was sind Isolationsebenen, welche gibt es, was ist Serialisierbarkeit?
 - Wie erreiche ich Serialisierbarkeit? Will ich das immer?
 - Was sind Sperren und Sperrprotokolle wie 2PL?
 - Wie setze ich das alles in SQL um?

Literatur: CompleteBook Chap 6.6, Chap 7; Biberbuch Kap 12



Kapitel 9: Transaktionen, Integrität und Trigger

9.1 Transaktionen

9.1.1 Transaktionsbegriff

9.1.2 Probleme im Mehrbenutzerbetrieb

9.1.3 Serialisierbarkeit

9.1.4 Sperrprotokolle zur Synchronisation

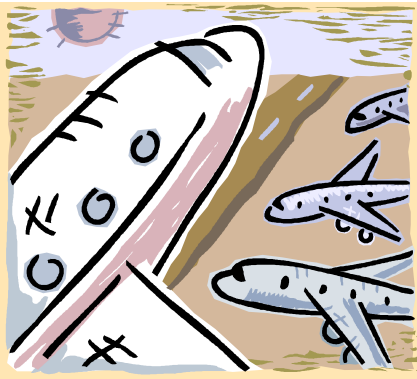
9.1.5 Transaktionen in SQL-DBMS

9.2 Integritätsbedingungen

9.3 Trigger

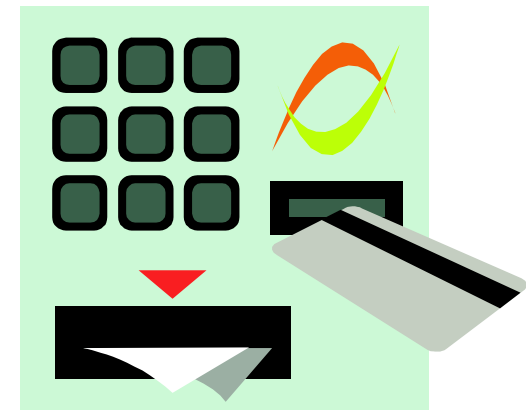


Beispielszenarien für Transaktionen



- ◆ Platzreservierung für Flüge gleichzeitig aus vielen Reisebüros
→ Platz könnte mehrfach verkauft werden, wenn mehrere Reisebüros den Platz als verfügbar identifizieren

- ◆ überschneidende Kontooperationen einer Bank
→ Konten könnten falsche Salden enthalten, wenn sich mehrere Überweisungen überschneiden
- ◆ statistische Datenbankoperationen
→ Ergebnisse sind verfälscht, wenn während der Berechnung Daten geändert werden

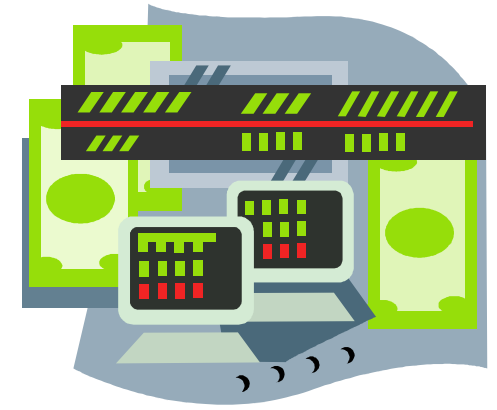




Der Transaktionsbegriff

◆ Definition **Transaktion**

Eine **Transaktion** ist eine Folge von Operationen (Aktionen), die die Datenbank von einem konsistenten Zustand in einen (eventuell veränderten) konsistenten Zustand überführt, wobei das ACID-Prinzip eingehalten werden muss.



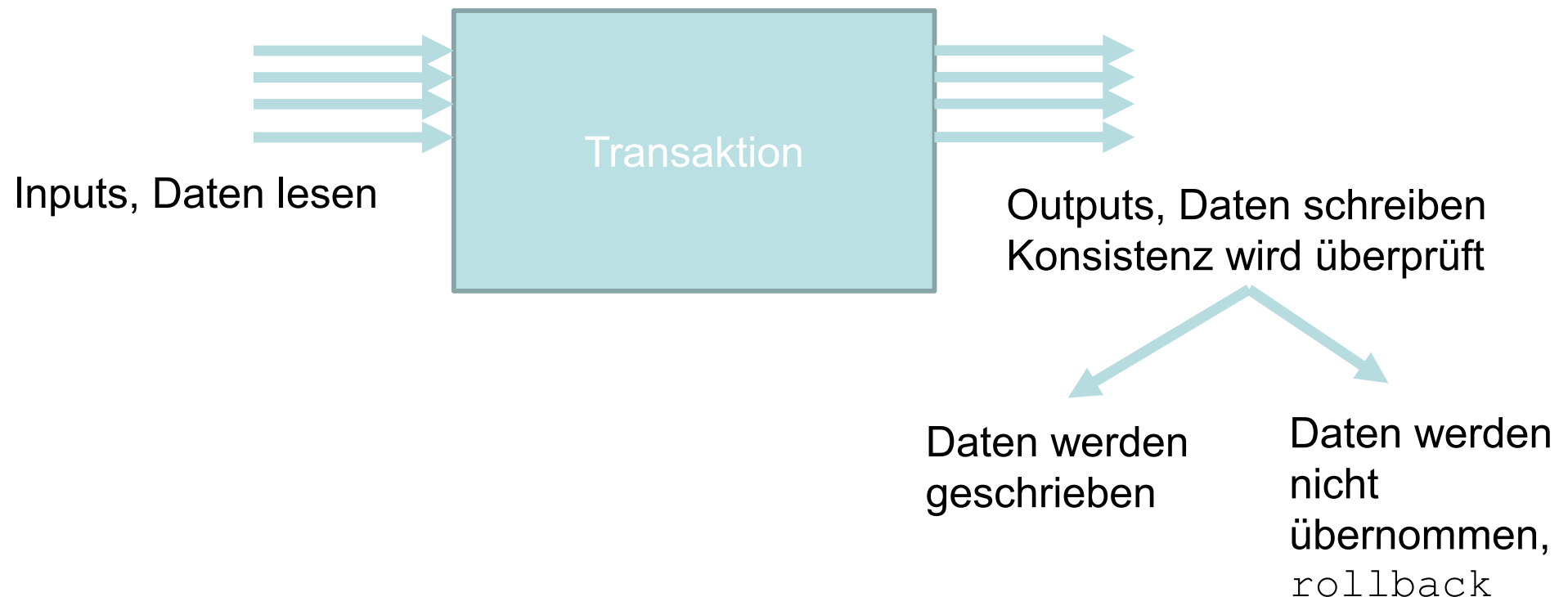
◆ Aspekte:

- **Semantische Integrität**: Korrekter (konsistenter) DB-Zustand nach Ende der Transaktion
- **Ablaufintegrität**: Fehler durch „gleichzeitigen“ Zugriff mehrerer Benutzer auf dieselben Daten vermeiden



Konsistenz einer Transaktion

- ♦ Operationen einer Transaktion können entweder Daten lesen (z.B. durch `SELECT`) oder Daten schreiben (z.B. durch `INSERT`, `UPDATE`, `DELETE`).





Zwei Gesetze der Nebenläufigkeit

- ◆ Nebenläufige oder gleichzeitige Ausführung von Aufgaben soll nicht dazu führen, dass Programme fehlerhaft ausgeführt werden. (Isolation in ACID)
 - Befinden sich alle Daten in einer zentralen Quelle, sind verfügbar von einer zentralen Recheneinheit aus und benötigen die Anwendungen nur eine sehr kurze Zeit, ist das Problem der Nebenläufigkeit leicht durch sequenzielle Ausführung zu lösen.
- ◆ Die nebenläufige Ausführung von Aufgaben soll nicht entscheidend langsamer ausgeführt werden als eine sequentielle Ausführung.



ACID-Eigenschaften

Atomicity (Atomarität):

Transaktion wird entweder ganz oder gar nicht ausgeführt

Consistency (Konsistenz oder auch Integritätserhaltung):

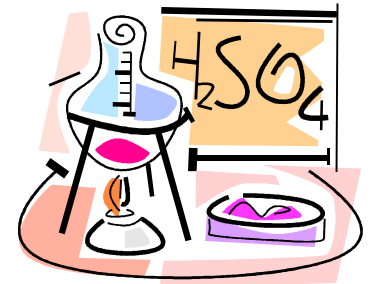
Datenbank ist vor Beginn und nach Beendigung einer Transaktion jeweils in einem konsistenten Zustand

Isolation (Isolation):

Nutzer, der mit einer Datenbank arbeitet, sollte den Eindruck haben, dass er mit dieser Datenbank alleine arbeitet

Durability (Dauerhaftigkeit / Persistenz):

nach erfolgreichem Abschluss einer Transaktion muss das Ergebnis dieser Transaktion „dauerhaft“ in der Datenbank gespeichert werden





Kommandos zur Transaktionssteuerung

- ◆ **BOT** (Begin-of-Transaction)
 - Beginn einer Transaktion: in SQL implizit!

- ◆ **commit:**
 - die Transaktion soll erfolgreich beendet werden (und die nächste implizit gestartet werden)

- ◆ **abort:**
 - die Transaktion soll abgebrochen werden (und die nächste implizit gestartet werden)





Integritätsverletzung in Transaktionen

◆ Beispiel einer Transaktion T:

- Tabelle KONTEN (KontoNr, Stand)
- Übertragung eines Betrages 200€ von Konto K1 auf ein anderes Konto K2
- Bedingung: Summe der Kontostände aller Konten bleibt konstant

◆ Realisierung der Transaktion T in SQL

- als Sequenz (zweier) elementarer Änderungen:

```
update KONTEN set Stand = Stand - 200 where KontoNr = K1  
update KONTEN set Stand = Stand + 200 where KontoNr = K2
```

➔ Bedingung ist zwischen den einzelnen Änderungsschritten nicht unbedingt erfüllt!



Vereinfachtes Modell für Transaktion

◆ Repräsentation von Datenbankänderungen einer Transaktion

- **read** (A, x) : weise den Wert des DB-Objektes A der Variablen x zu
- **write** (x, A) : speichere den Wert der Variablen x im DB-Objekt A

◆ Beispiel unserer Transaktion T:

```
read(StandKontoK1, x); x := x - 200; write(x, StandKontoK1);  
read(StandKontoK2, y); y := y + 200; write(y, StandKontoK2);  
commit
```

◆ Beispiel einer weiteren Transaktion S, die 100€ von K1 auf K3 überträgt:

```
read(StandKontoK1, u); u := u - 100; write(u, StandKontoK1);  
read(StandKontoK3, v); v := v + 100; write(v, StandKontoK3);  
commit
```



Ausführungsvarianten für zwei Transaktionen S, T

♦ Serielle Ausführung von S vor T:

S	T
read (StandKontoK1,u);	
u := u - 100;	
write (u,StandKontoK1);	
read (StandKontoK3,v);	
v := v + 100;	
write (v,StandKontoK3);	
commit	
	read (StandKontoK1,x);
	x := x - 200;
	write (x,StandKontoK1);
	read (StandKontoK2,y);
	y := y + 200;
	write (y,StandKontoK2);
	commit

♦ Gemischte“ Ausführung, etwa abwechselnd Schritte von S und T

S	T
read (StandKontoK1,u);	
	read (StandKontoK1,x);
u := u - 100;	
	x := x - 200;
write (u,StandKontoK1);	
	write (x,StandKontoK1);
read (StandKontoK3,v);	
	read (StandKontoK2,y);
v := v + 100;	
	y := y + 200;
write (v,StandKontoK3);	
	write (y,StandKontoK2);
Commit	
	commit



Kapitel 9: Transaktionen, Integrität und Trigger

9.1 Transaktionen

9.1.1 Transaktionsbegriff

9.1.2 Probleme im Mehrbenutzerbetrieb

9.1.3 Serialisierbarkeit

9.1.4 Sperrprotokolle zur Synchronisation

9.1.5 Transaktionen in SQL-DBMS

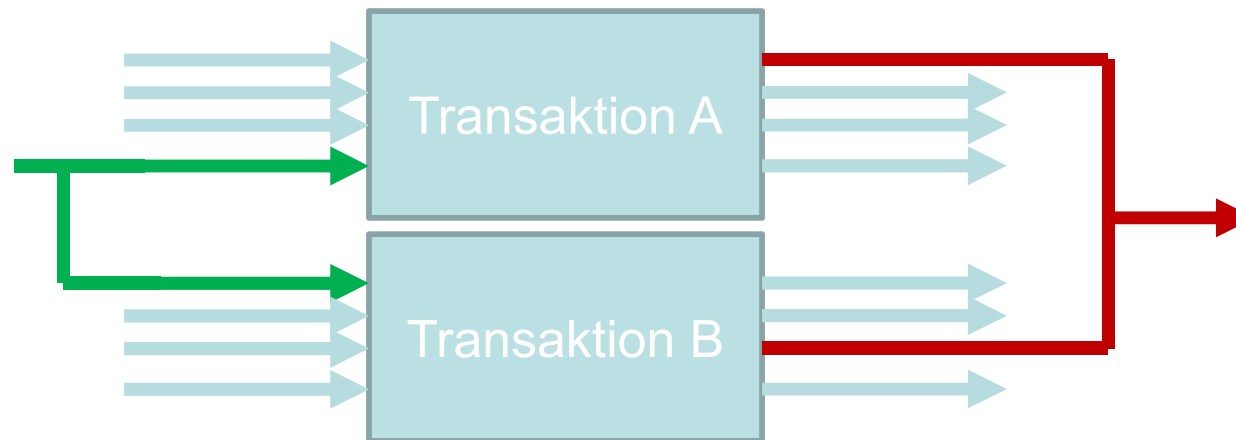
9.2 Integritätsbedingungen

9.3 Trigger



Nebenläufigkeit

- ♦ Greifen zwei Transaktionen gleichzeitig **nur lesend** auf ein Objekt zu, kann die Konsistenz nicht verletzt werden, da sich der Zustand des Objekts nicht verändert.
- ♦ Greifen zwei Objekte **schreibend** auf dasselbe Objekt zu, kann das zur Verletzung des *Isolation*-Prinzips führen.



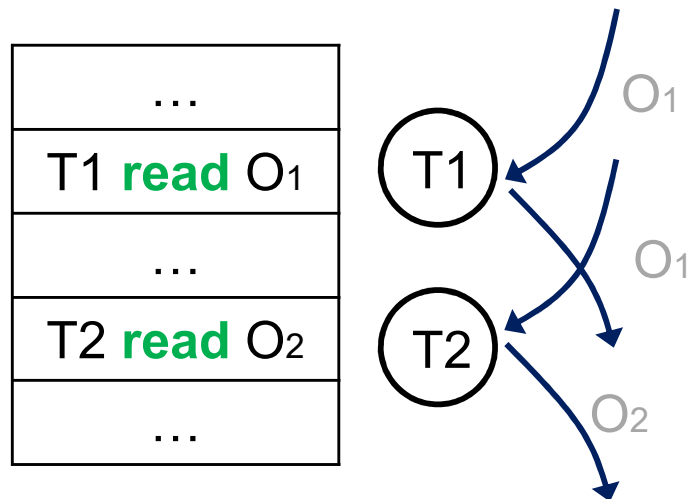
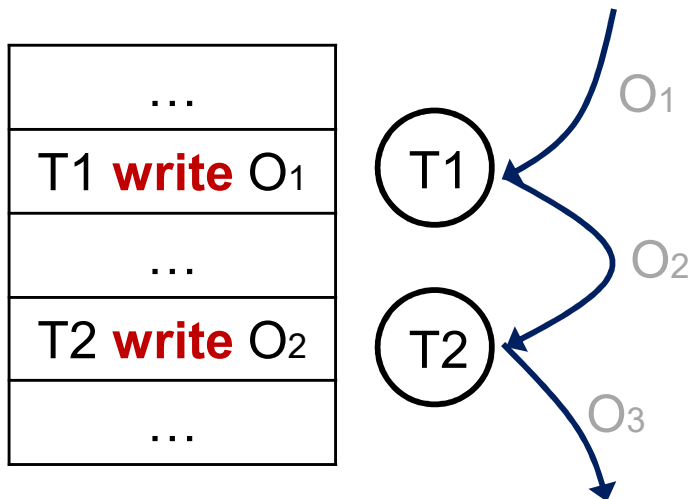
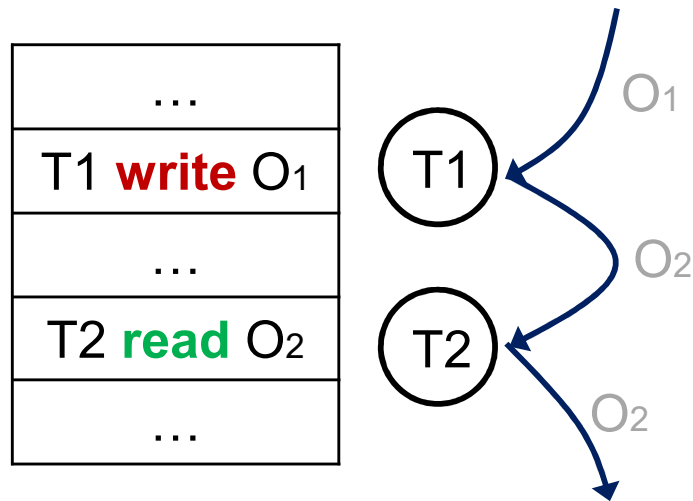
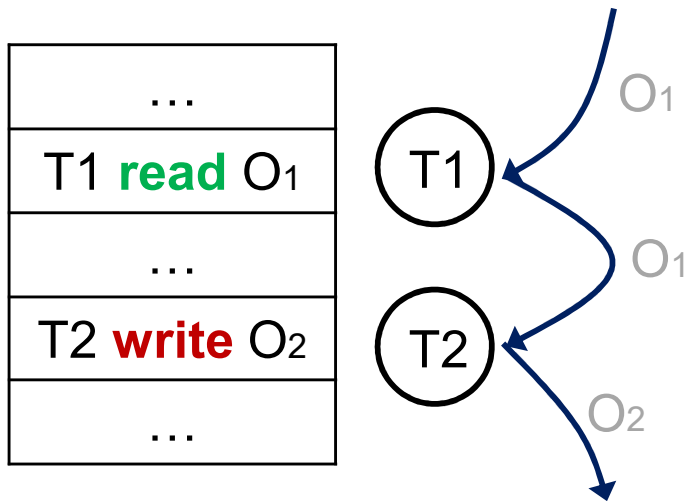


Statisch vs. Dynamische Allokation

- ◆ **Statische Allokation** bedeutet, dass eine Transaktion **zur Designzeit** explizit angibt, auf welche Objekte sie lesend und schreibend zugreift.
 - z.B. eine Kontobuchung greift auf ein beliebiges Konto schreibend zu, daher muss die gesamte Tabelle Konto allokiert werden. → sehr pessimistisch
- ◆ Ein Transaktionsmanager kann dann sehr leicht überprüfen, ob es zu Konflikten durch die nebenläufige Ausführung mit einer anderen Transaktion kommen kann und führt diese dann sequenziell aus.
- ◆ Bei der **dynamischen Allokation** werden Objekte **zur Laufzeit** allokiert
 - z.B. eine Kontobuchung greift auf ein Konto 12345 zu und nur dieses Konto wird allokiert. → hohe Nebenläufigkeit, aber schwer zu berechnen.



Abhängigkeitsgraph

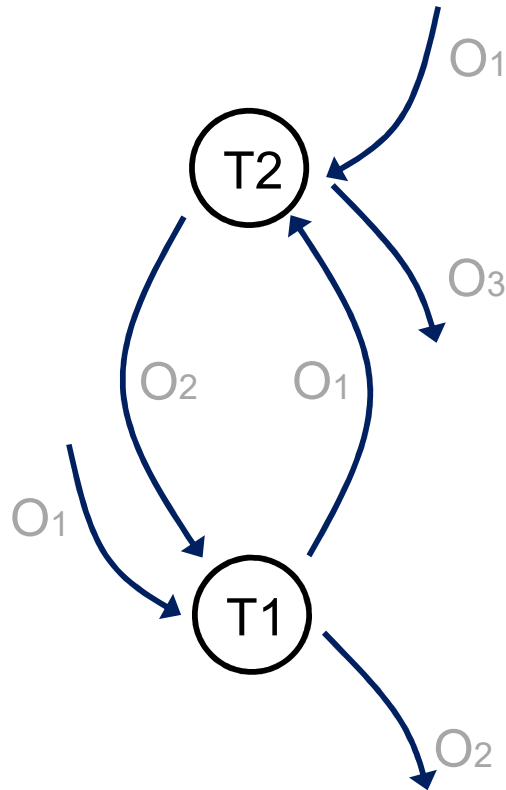


- ◆ Schreibende Zugriffe auf Daten erzeugen Abhängigkeiten in der nebenläufigen Ausführung von unterschiedlichen Transaktionen.
- ◆ Zyklfreiheit bedeutet sequentielle Ausführung ist möglich, Isolation gewährleistet



write → write Abhängigkeit und Lost Update

...
T2 read O ₁
...
T1 write O ₂
...
T2 write O ₃
...



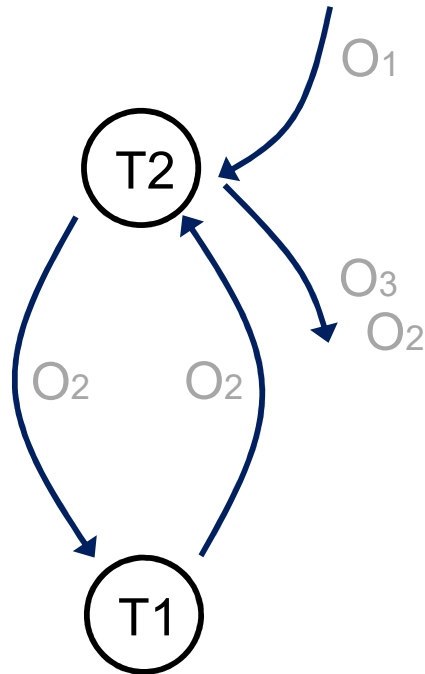
T1	T2	A
read (A, x);		10
	read (A, y);	10
x := x + 1;		10
	y := y + 1;	10
write (x, A);		11
	write (y, A);	11

- ♦ Eine Zyklus im Abhängigkeitsgraphen verursacht durch eine **write** → **write** Abhängigkeit **kann** zu einem s.g. **Lost Update** führen.



write → read Abhängigkeit und Dirty Read

...
T2 write O ₂
...
T1 read O ₂
...
T2 write O ₃
...



T1	T2
read (A, x);	
x := x / 100;	
write (x, A);	
	read (A, x);
	read (B, y);
	y := y + x;
	write (y, B);
	commit ;
abort ;	

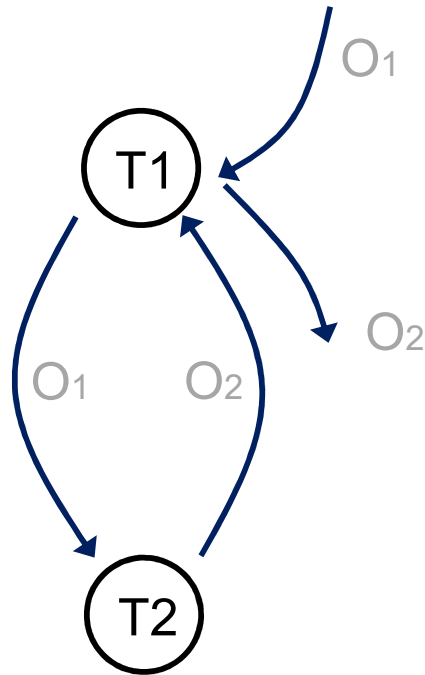
O₂

- ♦ Eine Zyklus im Abhängigkeitsgraphen verursacht durch eine **write** → **read** Abhängigkeit **kann** zu einem s.g. **Dirty Read** führen.



read → write Abhängigkeit und Unrepeatable Read

...
T1 read O ₁
...
T2 write O ₂
...
T1 read O ₂
...



- ◆ Eine Zyklus im Abhängigkeitsgraphen verursacht durch eine **read** → **write** Abhängigkeit **kann** zu einem s.g. **Unrepeatable Read** führen.

T1	T2
read (A, x);	
	read (A, y);
	y := y / 2;
	write (y, A);
	read (C, z);
	z := z + y;
	write (z, C);
	commit ;
read (B, y);	
x := x + y;	
read (C, z);	
x := x + z;	
commit ;	

O₂



Phantom-Problem

- ◆ Nicht nur Tupel in der Datenbank können Objekte sein, sondern auch Indizes.
- ◆ Beispiel für einen Spezialfall des Unrepeatable Reads auf einem Index. Hier wird der Index während der Ausführung von T1 verändert und damit ist die Berechnung des Bonus zu diesem Zeitpunkt nicht mehr korrekt.

T1	T2
select count (*) into X from Kunde	
	insert into Kunde values ('Meier', 0, ...)
	commit;
update Kunde set Bonus = Bonus +10000/X;	
commit;	



Zusammenfassung: Probleme im Mehrbenutzerbetrieb

- ◆ **Lost Update:** Verlorengegangene Änderungen
- ◆ **Dirty Read:** Abhängigkeiten von nicht freigegebenen Daten
- ◆ **Phantom-Problem:** Gibt es den Datensatz, oder nicht?
- ◆ **Nonrepeatable Read:** Inkonsistentes Lesen



Kapitel 9: Transaktionen, Integrität und Trigger

9.1 Transaktionen

9.1.1 Transaktionsbegriff

9.1.2 Probleme im Mehrbenutzerbetrieb

9.1.3 Serialisierbarkeit

9.1.4 Sperrprotokolle zur Synchronisation

9.1.5 Transaktionen in SQL-DBMS

9.2 Integritätsbedingungen

9.3 Trigger



Beispiele für verschränkte Ausführungen

♦ Zwei Transaktionen

- T_1 : `read(A,x); x:=x-10; write(x,A); read(B,y); y:=y+10; write(y,B);`
- T_2 : `read(B,y); y:=y-20; write(y,B); read(C,z); z:=z+20; write(z,C);`

♦ Beispiele für verschränkte Ausführungen

Ausführung 1	
T1	T2
<code>read(A,x);</code>	
<code>x := x-10;</code>	
<code>write(x,A);</code>	
<code>read(B,y);</code>	
<code>y := y+10;</code>	
<code>write(y,B);</code>	
	<code>read(B,y);</code>
	<code>y := y-20;</code>
	<code>write(y,B);</code>
	<code>read(C,z);</code>
	<code>z := z+20;</code>
	<code>write(z,C);</code>



Beispiele für verschränkte Ausführungen

◆ Zwei Transaktionen

- T_1 : `read(A,x); x:=x-10; write(x,A); read(B,y); y:=y+10; write(y,B);`
- T_2 : `read(B,y); y:=y-20; write(y,B); read(C,z); z:=z+20; write(z,C);`

◆ Beispiele für verschränkte Ausführungen

Ausführung 1		Ausführung 2		Ausführung 3	
T1	T2	T1	T2	T1	T2
read (A,x);		read (A,x);		read (A,x);	
x := x-10;			read (B,y);	x := x-10;	
write (x,A);		x := x-10;			read (B,y);
read (B,y);			y := y-20;	write (x,A);	
y := y+10;		write (x,A);			y := y-20;
write (y,B);			write (y,B);	read (B,y);	
	read (B,y);	read (B,y);			write (y,B);
	y := y-20;		read (C,z);	y := y+10;	
	write (y,B);	y := y+10;			read (C,z);
	read (C,z);		z := z+20;	write (y,B);	
	z := z+20;	write (y,B);			z := z+20;
	write (z,C);		write (z,C);		write (z,C);



Serialisierbarkeit (1)

♦ Effekt der unterschiedlichen Ausführungen

	A	B	C	A+B+C
initialer Wert	10	10	10	30
nach Ausführung 1	0	0	30	30
nach Ausführung 2	0	0	30	30
nach Ausführung 3	0	20	30	50

♦ Definition **Serialisierbarkeit**

Eine verschränkte Ausführung mehrerer Transaktionen heißt **serialisierbar**, wenn ihr Effekt identisch zum Effekt einer (beliebig gewählten) seriellen Ausführung dieser Transaktionen ist.



Serialisierbarkeit (2) – Read/Write Modell

- ◆ Das Read/Write-Modell

Transaktion T ist eine endliche Folge von Operationen (Schritten) p_i der Form $r(x_i)$ oder $w(x_i)$:

$$T = p_1 p_2 p_3 \dots p_n \text{ mit } p_i \in \{r(x_i), w(x_i)\}$$

- ◆ **Vollständige Transaktion T** hat als letzten Schritt entweder einen Abbruch a oder ein Commit c :

$$T = p_1 \dots p_n a$$

oder

$$T = p_1 \dots p_n c.$$



Serialisierbarkeit (3) - Schedule

- ◆ Ein **vollständiger Schedule** ist eine Folge von DB-Operationen, so dass alle Operationen zu vollständigen Transaktionen gehören und alle Operationen dieser Transaktionen im Schedule in derselben relativen Reihenfolge auftreten wie in der Transaktion.
- ◆ Ein **Schedule** ist ein Präfix eines vollständigen Schedules.

- ◆ Beispiel:

$r_1(x) \ r_2(x) \ w_1(x) \ r_2(y) \ a_1 \ w_2(y) \ c_2$



Schedule



Vollständiger Schedule



Serialisierbarkeit (4) – Serieller Schedule

- ◆ Ein **serieller Schedule** s für T_1, \dots, T_n ist ein vollständiger Schedule der folgenden Form:

$$s := T_{\rho(1)}, \dots, T_{\rho(n)} \text{ für eine Permutation } \rho \text{ von } \{1, \dots, n\}$$

- ◆ Beispiel: serielle Schedules für zwei Transaktionen
 $T_1 := r_1(x) w_1(x) c_1$ und $T_2 := r_2(x) w_2(x) c_2$:

$$s_1 := \underbrace{r_1(x) w_1(x) c_1}_{T_1} \underbrace{r_2(x) w_2(x) c_2}_{T_2}$$

$$s_2 := \underbrace{r_2(x) w_2(x) c_2}_{T_2} \underbrace{r_1(x) w_1(x) c_1}_{T_1}$$



Serialisierbarkeit (5) – Korrektheitskriterium

- ♦ Ein Schedule s ist **korrekt**, wenn der Effekt des Schedules s (Ergebnis der Ausführung des Schedules) äquivalent dem Effekt eines (beliebigen) seriellen Schedules s' bzgl. derselben Menge von Transaktionen ist (in Zeichen $s \approx s'$).
- ♦ Ist ein Schedule s äquivalent zu einem seriellen Schedule s' , dann ist s **serialisierbar** (zu s').
- ♦ Frage: wie stellt man die Serialisierbarkeit bei maximaler Parallelität sicher?
- ➔ Optimistische Verfahren (Probieren und ggf. rückgängig machen)
- ➔ Pessimistische Verfahren (Sperrprotokolle verwenden)



Kapitel 9: Transaktionen, Integrität und Trigger

9.1 Transaktionen

9.1.1 Transaktionsbegriff

9.1.2 Probleme im Mehrbenutzerbetrieb

9.1.3 Serialisierbarkeit

9.1.4 Sperrprotokolle zur Synchronisation

9.1.5 Transaktionen in SQL-DBMS

9.2 Integritätsbedingungen

9.3 Trigger



Sperrprotokolle

- ◆ Sichern der Serialisierbarkeit durch exklusiven Zugriff auf Objekte (Synchronisation der Zugriffe)
- ◆ Implementierung über Sperren und Sperrprotokolle
- ◆ Sperrprotokoll garantiert Serialisierbarkeit ohne zusätzliche Tests!



Sperrmodelle (elementare Sperren)

- ◆ Schreib- und Lesesperren in folgender Notation:
 - $rl(x)$: Lesesperre (engl. read lock) auf einem Objekt x
 - $wl(x)$: Schreibsperre (engl. write lock) auf Objekt x
 - Entsperren $ru(x)$ und $wu(x)$, oft zusammengefasst $u(x)$ für engl. Unlock
- ◆ Kompatibilitätsmatrix für elementare Sperren

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	ok	-
$wl_j(x)$	-	-



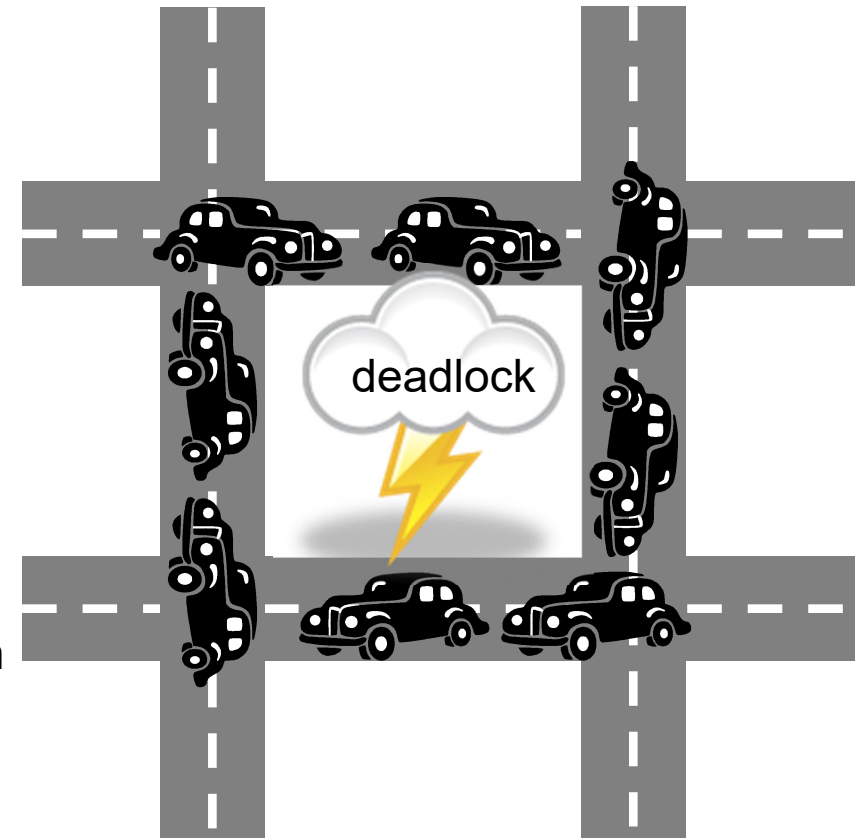
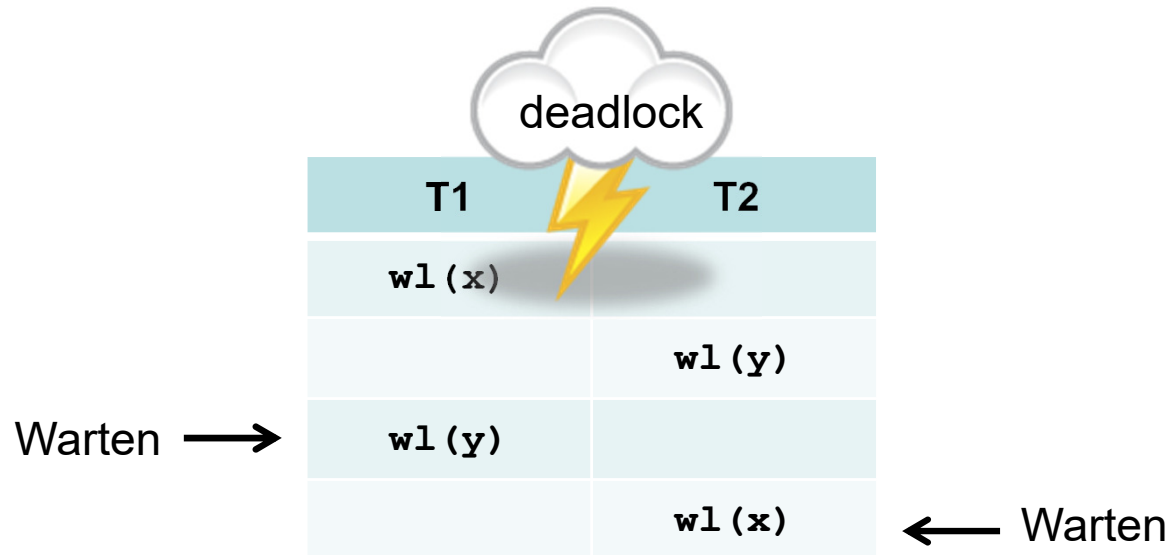
Sperrdisziplin

- ◆ Schreibzugriff $w(x)$ nur nach Setzen einer Schreibsperre $wl(x)$ möglich
- ◆ Lesezugriffe $r(x)$ nur nach $rl(x)$ oder $wl(x)$ erlaubt
- ◆ nur Objekte sperren, die nicht bereits von einer anderen Transaktion gesperrt sind
- ◆ Sperren derselbe Art werden maximal einmal gesetzt, d.h. genauer
 - nach $rl(x)$ nur noch $wl(x)$ erlaubt, danach auf x keine Sperre mehr
 - nach $u(x)$ durch T_i darf T_i kein erneutes $rl(x)$ oder $wl(x)$ ausführen
- ◆ vor einem **commit** müssen alle Sperren aufgehoben werden



Verklemmungen (deadlocks)

♦ Beispiel



♦ Alternativen

- Verklemmungen werden erkannt und **beseitigt**
- Verklemmungen werden von vornherein **vermieden**

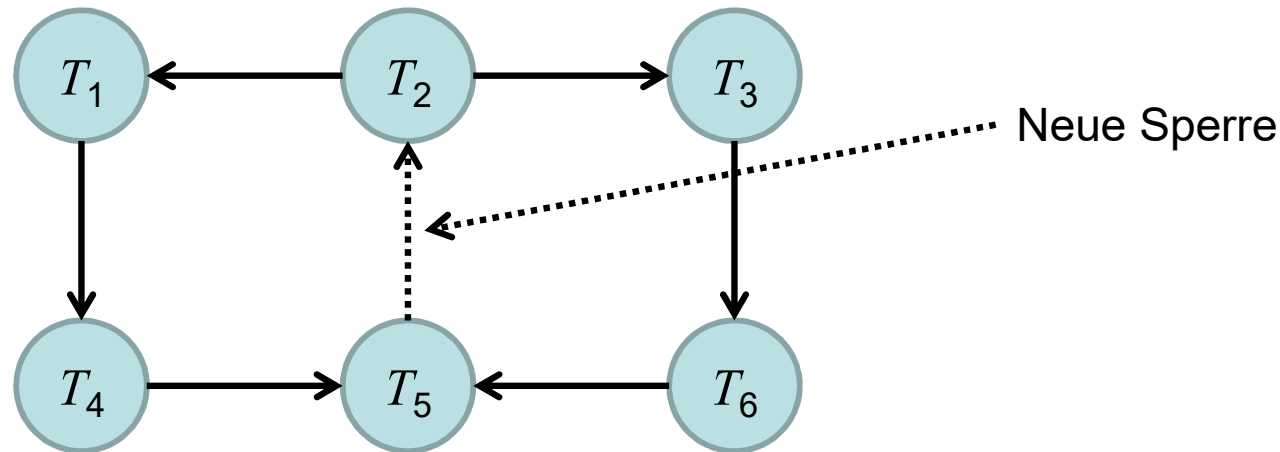


Quelle: <http://minutillo.com/steve/weblog/2003/1/21/deadlock/>



Verklemmungserkennung und -auflösung

♦ Wartegraph



♦ Auflösen durch **Abbruch einer Transaktion**, Kriterien:

- Anzahl der aufgebrochenen Zyklen
- Länge einer Transaktion
- Rücksetzaufwand einer Transaktion
- Wichtigkeit einer Transaktion
- . . .



Notwendigkeit von Sperrprotokollen

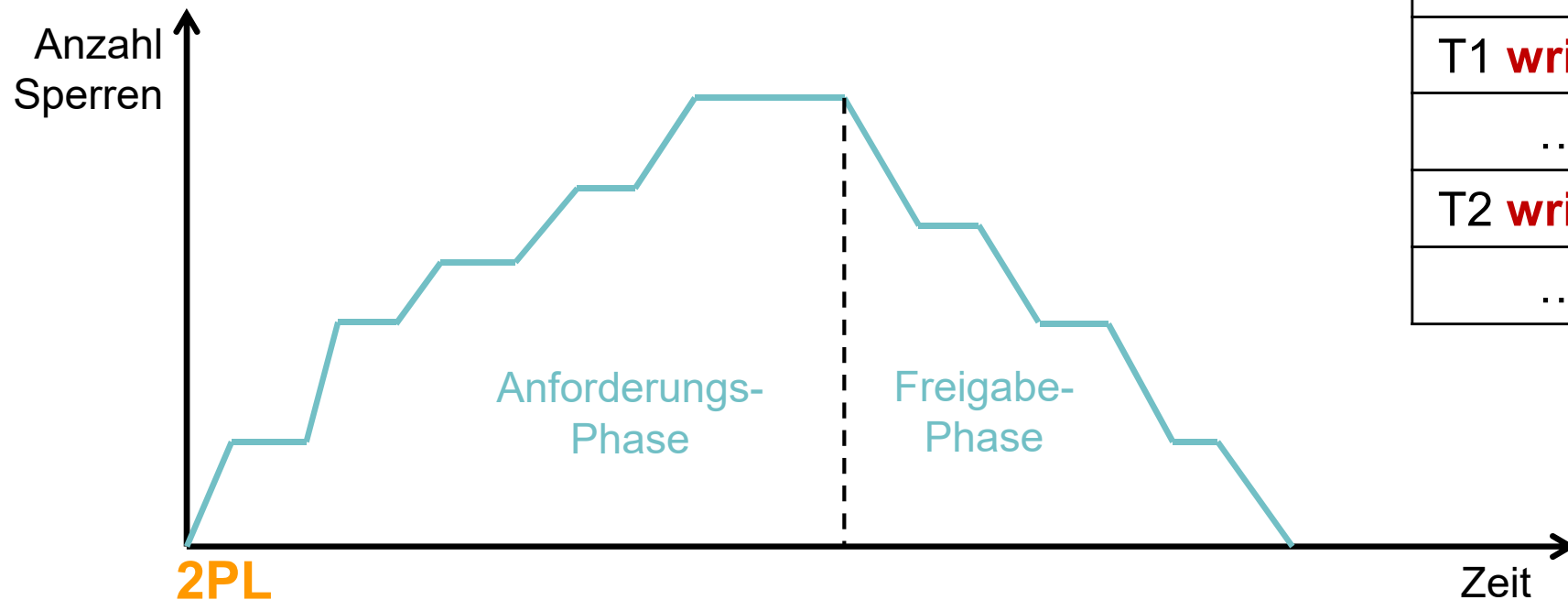
- ◆ Beispiel: Hier wird brav gesperrt (w_l) und sperren nach dem Zugriff (w) wieder freigegeben (u). Hilft aber nichts, denn T1 überschreibt y und T2 überschreibt x .

T1	T2
$w_l(x)$	
$w(x)$	
$u(x)$	
	$w_l(x)$
	$w(x)$
	$u(x)$
	$w_l(y)$
	$w(y)$
	$u(y)$
$w_l(y)$	
$w(y)$	
$u(y)$	



Zwei-Phasen-Sperr-Protokoll

- ◆ Zwei-Phasen-Sperr-Protokoll (2-phase-locking, 2PL)



...
T2 read O ₁
...
T1 write O ₂
...
T2 write O ₃
...

- ◆ Wenn nachdem eine Sperre freigegeben wurde, keine erneute Sperre angefordert wird, können Zyklen im Abhängigkeitsgraphen nicht zu Isolationsproblemen führen, da die Objekte ja geblockt sind.



Deadlocks und Schneeballeffekt

- ◆ Bei der stufenweisen Anforderungsphase des Zwei-Phasen-Sperr-Protokolls kann es durch Zyklen im Abhängigkeitsgraphen zu Deadlocks kommen, da dann Transaktionen gegenseitig auf die Freigabe ihrer Sperren warten.
 - T1 wartet auf T2 zur Freigabe von O, T2 wartet auf T1 zur Freigabe von U
- ◆ Bei der stufenweisen Freigabe von Sperren vor dem Ende der Transaktion kann es zu kaskadierendem Zurücksetzen kommen, wenn eine Transaktion rückgängig gemacht wird, aber auf den Freigaben bereits weiter gearbeitet wurde.
 - T2 verändert O, gibt O frei, T1 liest O, T2 wird rückgängig gemacht, T1 damit ungültig.

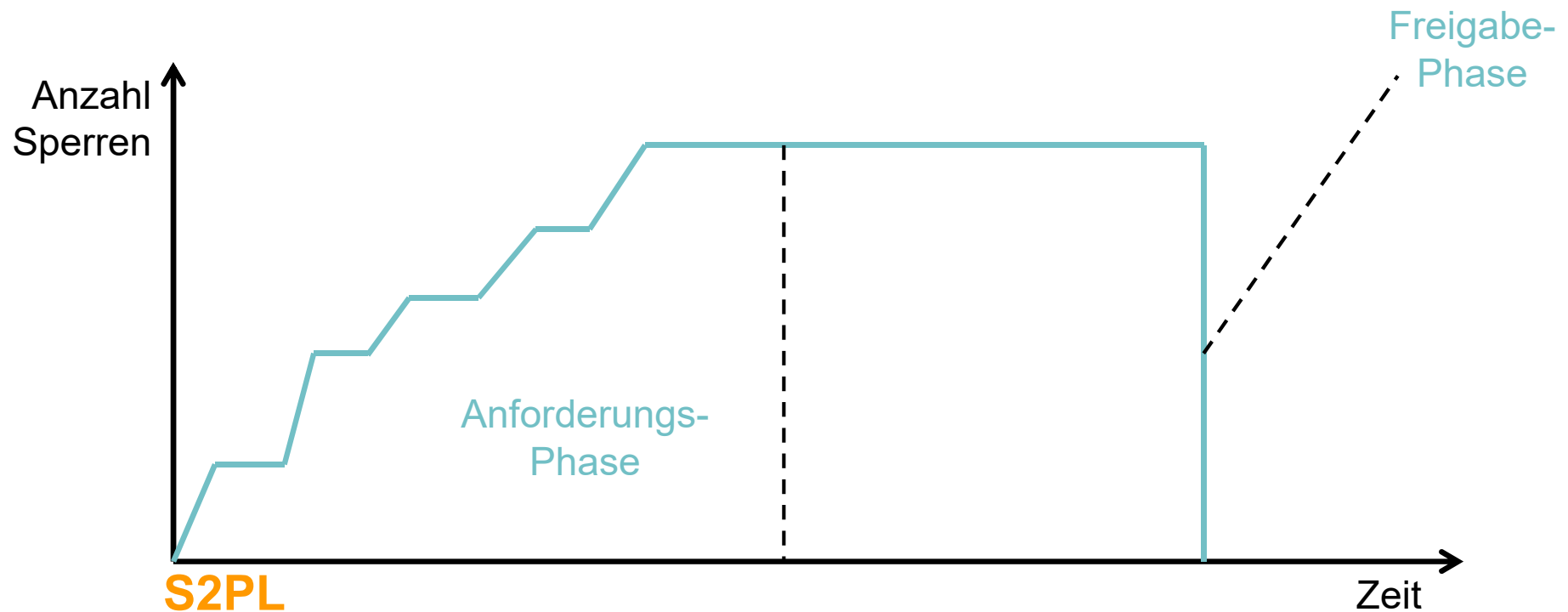
...
T2 read O
T1 read U
T1 write O
T2 write U
...

...
T2 write O ₂
...
T1 read O ₂
...
T2 abort
...



Striktes Zwei-Phasen-Sperr-Protokoll

- ♦ Striktes Zwei-Phasen-Sperr-Protokoll (strict 2-phase-locking, S2PL)

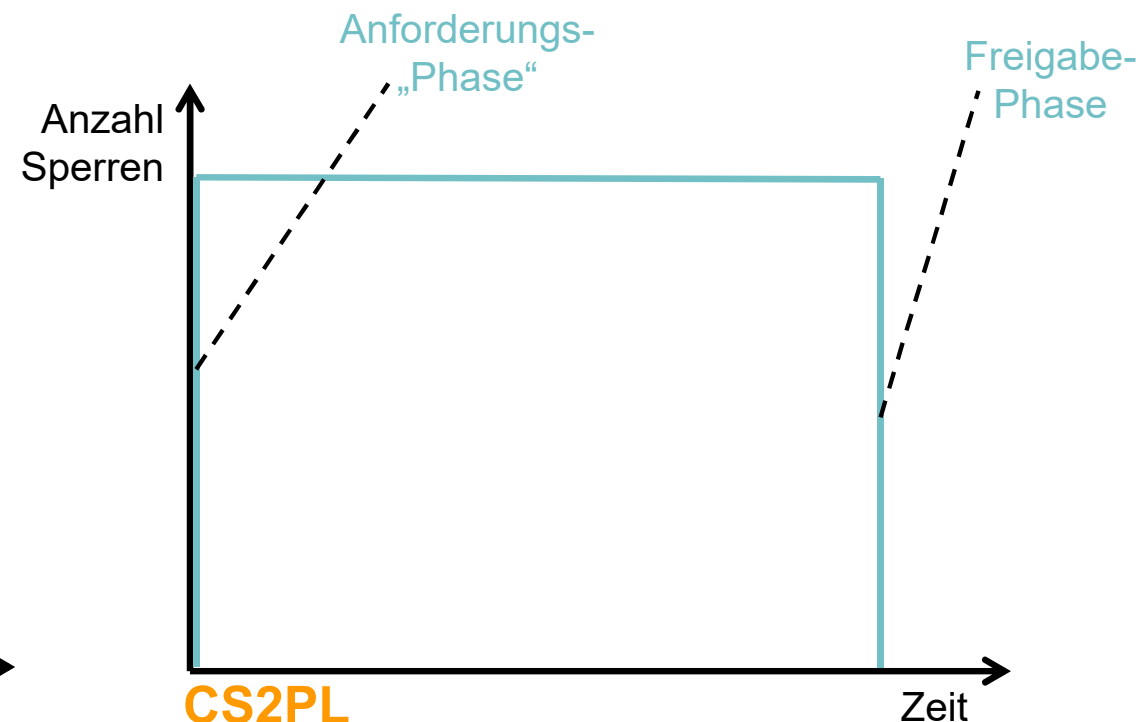
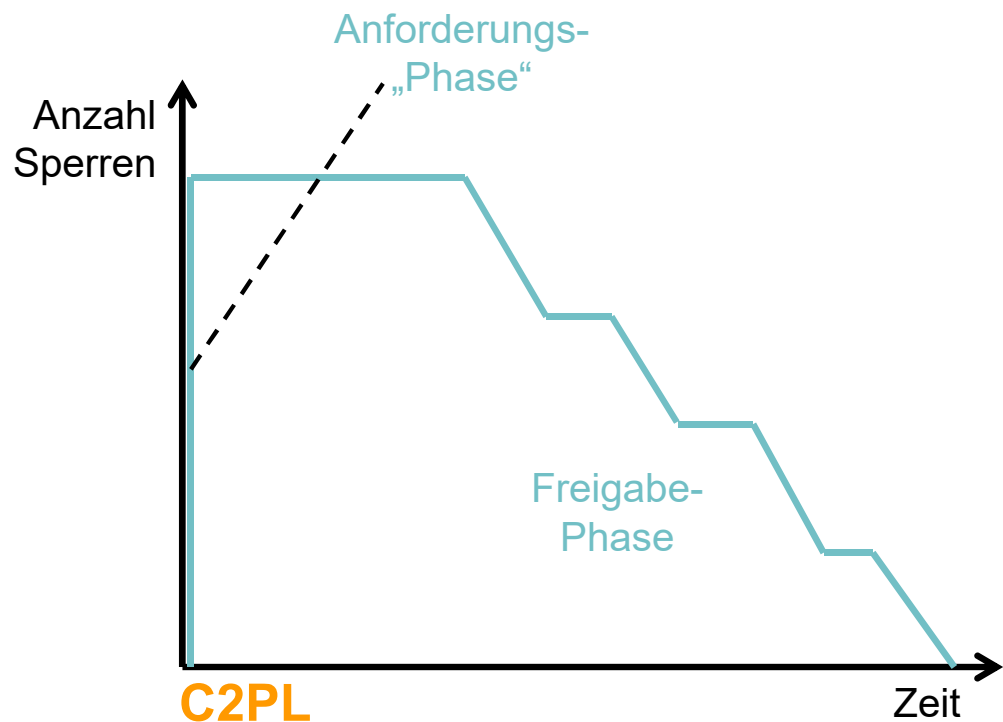


- ♦ Vermeidet kaskadierende Abbrüche durch Freigabe nach Abschluss der Transaktion, Deadlocks möglich



Konservatives Zwei-Phasen-Sperr-Protokoll

- ◆ Konservatives Zwei-Phasen-Sperr-Protokoll (conservative 2PL, **C2PL**)
- ◆ Konservatives striktes Zwei-Phasen-Sperr-Protokoll (**CS2PL**)



- ◆ Vermeiden Deadlocks, CS2PL resultiert aber meist in sequentieller Ausführung. Vorhersage aller Sperren oft unmöglich (vgl statische Allokation)



Weitere Lock-Level

- ◆ Komplexere Locks zur Steigerung der Parallelität möglich
 - Shared (Read) (S)
 - Update Lock (U)
 - Exclusive Lock (X)
- ◆ Häufig noch hierarchische Locks
 - Intent Shared (S)
 - Intent Exclusive (IX)
 - Shared with Intent Exclusive (SIX)
- ◆ Und einige weitere....

Requested mode	Existing granted mode					
	IS	S	U	IX	SIX	X
Intent shared (IS)	Yes	Yes	Yes	Yes	Yes	No
Shared (S)	Yes	Yes	Yes	No	No	No
Update (U)	Yes	Yes	No	No	No	No
Intent exclusive (IX)	Yes	No	No	Yes	No	No
Shared with intent exclusive (SIX)	Yes	No	No	No	No	No
Exclusive (X)	No	No	No	No	No	No

Lock-Kompatibilität in MS SQL Server [www.microsoft.com]



Kapitel 9: Transaktionen, Integrität und Trigger

9.1 Transaktionen

9.1.1 Transaktionsbegriff

9.1.2 Probleme im Mehrbenutzerbetrieb

9.1.3 Serialisierbarkeit

9.1.4 Sperrprotokolle zur Synchronisation

9.1.5 Transaktionen in SQL-DBMS

9.2 Integritätsbedingungen

9.3 Trigger



Isolationsebenen in SQL

- ◆ Steigerung der Performance: Aufweichung der Serialisierbarkeit

```
set transaction [ { read only | read write }, ]  
[isolation level { read uncommitted |  
                  read committed |  
                  repeatable read |  
                  serializable }, ]  
  
...
```

- ◆ Standard

```
set transaction read write,  
isolation level serializable
```



Bedeutung der Isolationsebenen (1)

◆ **read uncommitted**

- schwächste Stufe: Zugriff auf nicht geschriebene Daten, nur für **read only** Transaktionen
- statistische und ähnliche Transaktionen (ungefährer Überblick, nicht korrekte Werte)
- keine Sperren → effizient ausführbar, andere Transaktionen werden NICHT behindert

◆ **read committed**

- nur Lesen endgültig geschriebener Werte, aber *nonrepeatable read* möglich

◆ **repeatable read**

- kein *nonrepeatable read*, aber *Phantomproblem* kann auftreten













◆ **serializable**

- garantierte Serialisierbarkeit



Bedeutung der Isolationsebenen (2)

- ◆ Auftretende () und vermiedene () Probleme pro Isolationsebene

Isolationsebene	Dirty Read	Nonrepeatable Read	Lost Update	Phantom Read
read uncommitted				
read committed				
repeatable read				
serializable	