FFI / Datenbanksysteme

Transaktionen

Prof. Dr. Patrick Cato

Technische Hochschule Ingolstadt

Transaktionskonzept

Definition

Eine Transaktion ist eine **logische Verarbeitungseinheit** auf einer Datenbank, die eine oder mehrere Datenbankoperationen (Einfügen, Löschen, Ändern und /oder Suchen) umfasst. Eine Transaktion wird mit dem Befehl **commit** als gültig erklärt oder mit dem Befehl **rollback** rückgängig gemacht. Der Beginn einer Transaktion wird mit BOT (Begin of Transaction) gekennzeichnet.

Eigenschaften von Transaktionen

Übersicht

Datenbanksysteme müssen so implementiert werden, dass diese sogenannten ACID-Eigenschaften sichergestellt werden

A	Atomarität (atomicity)
C	Konsistenz (consistency)
I	Isolation (isolation)
D	Dauerhaftigkeit (durability)

Atomarität

Die Teilschritte einer Transaktion werden vom DBMS als eine unteilbare, atomare Einheit durchgeführt, d.h. alle oder gar keiner.

Merke: **„Alles oder nichts“.**

Beispiel: Werden die 50 € von Konto A abgebucht, so müssen sie auch Konto B gutgeschrieben werden; ein ”Verlust“ der 50€ ist ausgeschlossen. Fällt bspw. der Strom zwischen den beiden Buchungsvorgängen aus, so muss bei **Wiederanlauf des RDBMS** die Abbuchung von Konto A **automatisch rückgängig** gemacht werden

Konsistenz

Die Datenbank hat vor und nach der Transaktion stets einen konsistenten Zustand, d.h. alle Integritätsbedingungen des Datenbankschemas sind erfüllt.

Merke: „Daten bleiben konsistent“

Beispiel: Eine Integritätsbedingung könnte sein, dass kein Konto seinen Kreditrahmen überschreiten darf; wird diese Konsistenzbedingung für A verletzt, darf die Überweisung nicht durchgeführt werden

Isolation

Eine Transaktion läuft isoliert gegenüber dem Einfluss anderer Transaktionen ab, so als ob sie exklusiven Zugriff auf die Daten hätte. Eventuell wird die Transaktion vom DBMS abgebrochen, weil andernfalls ein unerlaubter Einfluss anderer Transaktionen erfolgt wäre.

Merke: „Eine Transaktion hat die Daten quasi allein“

Beispiel: Kontostand von Konto A darf nicht geschrieben werden, wenn er zwischenzeitlich durch eine andere Transaktion verändert wurde

Dauerhaftigkeit

Die Ergebnisse einer bestätigten Transaktion (Acknowledgment, ACK) sind dauerhaft gesichert, d.h. das DBMS garantiert, dass bei einem Fehler der bestätigte Zustand wiederhergestellt werden kann.

Merke: „Nichts geht verloren“

Beispiel: Fällt der Strom in der Bank aus, kurz nachdem unsere Überweisung durchgeführt wurde, müssen sich die 50€ noch immer auf Konto B befinden

Beispielszenarien

- Klassisch: Finanztransaktionen (überschneidende Kontooperationen einer Bank)
- Flug- und Hotelbuchungen: Platz könnte mehrfach verkauft werden, wenn mehrere Reisebüros den Platz als verfügbar identifizieren (heute überbuchen Airlines in der Regel, sodass keine harten ACID-Eigenschaften definiert sind)

Beispiel Banküberweisung - mit Transaktion

- Wir wollen eine Überweisung durchführen von Konto 1 auf Konto 2
- Die Daten der Konten werden in der Tabelle Konto aufgezeichnet

KtoNr	Saldo
1	100
2	100

```
begin transaction;
update Konto set Saldo = Saldo - 50 where KtoNr = 1;
update Konto set Saldo = Saldo + 50 where KtoNr = 2;
commit;
```

Beispiel: Überweisung von 50 Euro von Konto A nach Konto B

1. `begin of transaction`
2. `a = read(A)` Lese Kontostand von Konto A in Variable a
3. `a = a - 50` Reduziere den Wert von a um 50
4. `write(A,a)` Schreibe neuen Kontostand von A

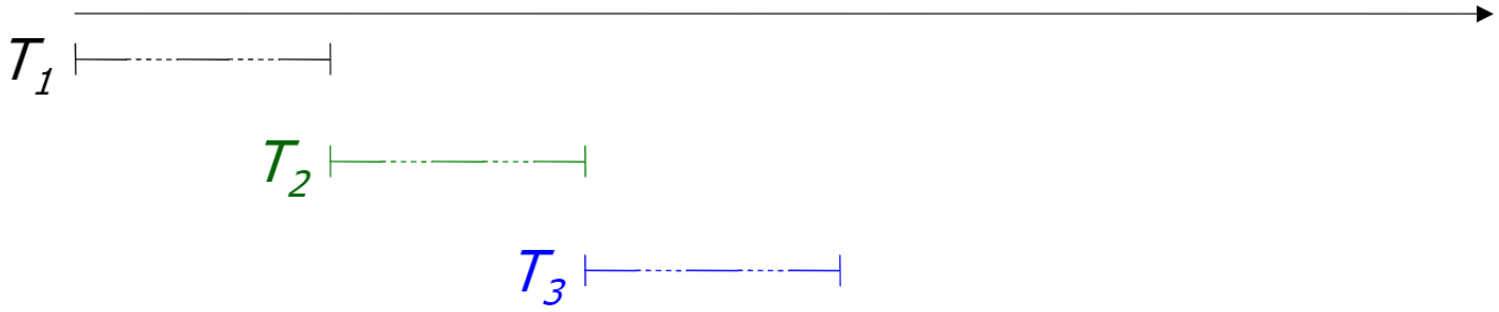
5. `b = read(B)` Lese Kontostand von Konto B in Variable b
6. `b = b + 50` Erhöhe den Wert von b um 50
7. `write(B,b)` Schreibe neuen Kontostand von B
8. `commit`

Transaktionsanfang und Transaktionsende

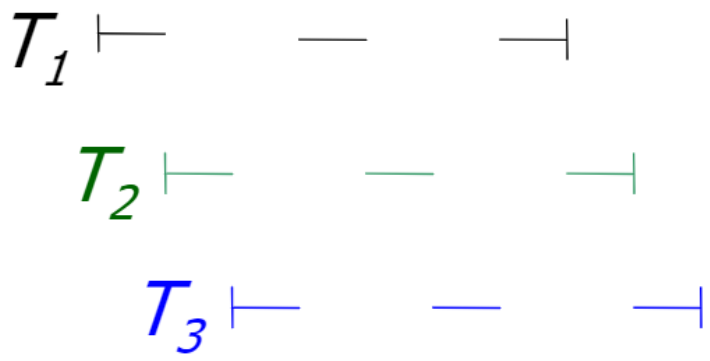
- begin of transaction / begin markiert den Anfang einer Transaktion
- commit markiert das Ende einer erfolgreichen Transaktion, deren Änderung festgeschrieben werden soll
- rollback / abort transaction markiert das Ende einer erfolglosen Transaktion, deren Änderungen rückgängig gemacht werden sollen

Mehrbenutzersynchronisation

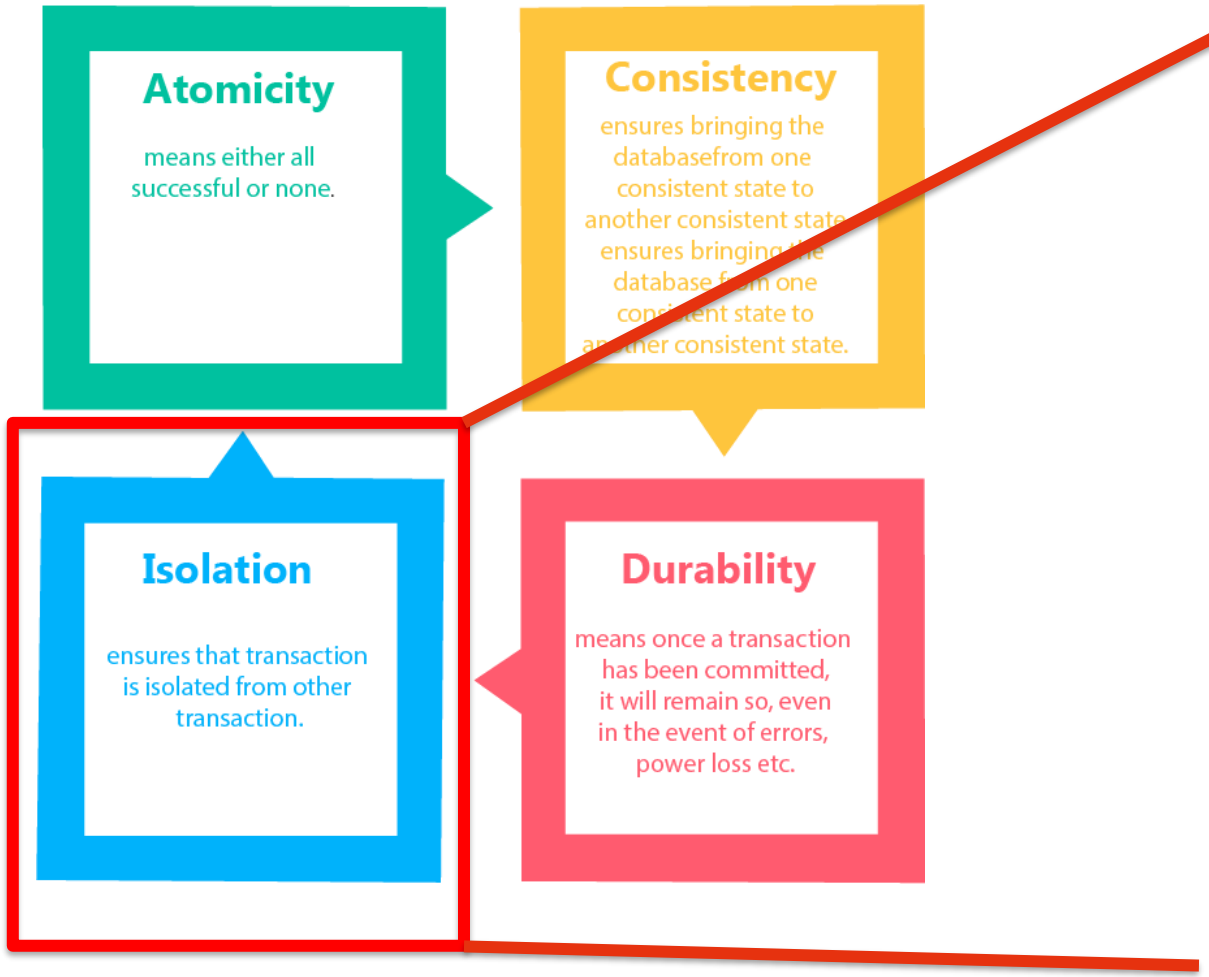
Einzelbetrieb



Mehrbenutzerbetrieb



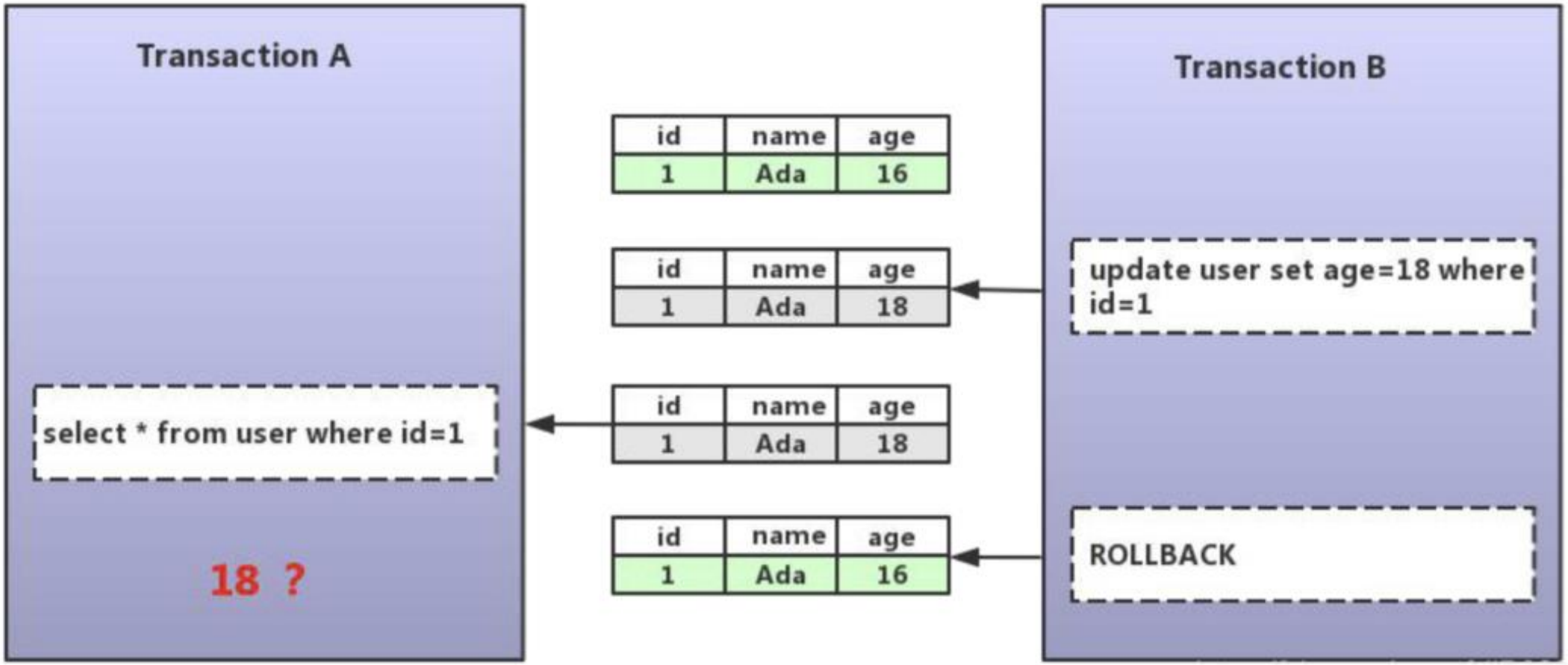
Problemraum des Mehrbenutzerbetriebs



Wenn Transaktionen nicht ausreichend isoliert sind kann es zu folgenden Problemen kommen:

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

Dirty Read



https://jennyttt.medium.com/dirty-read-non-repeatable-read-and-phantom-read-bd75dd69d03a

Dirty Read

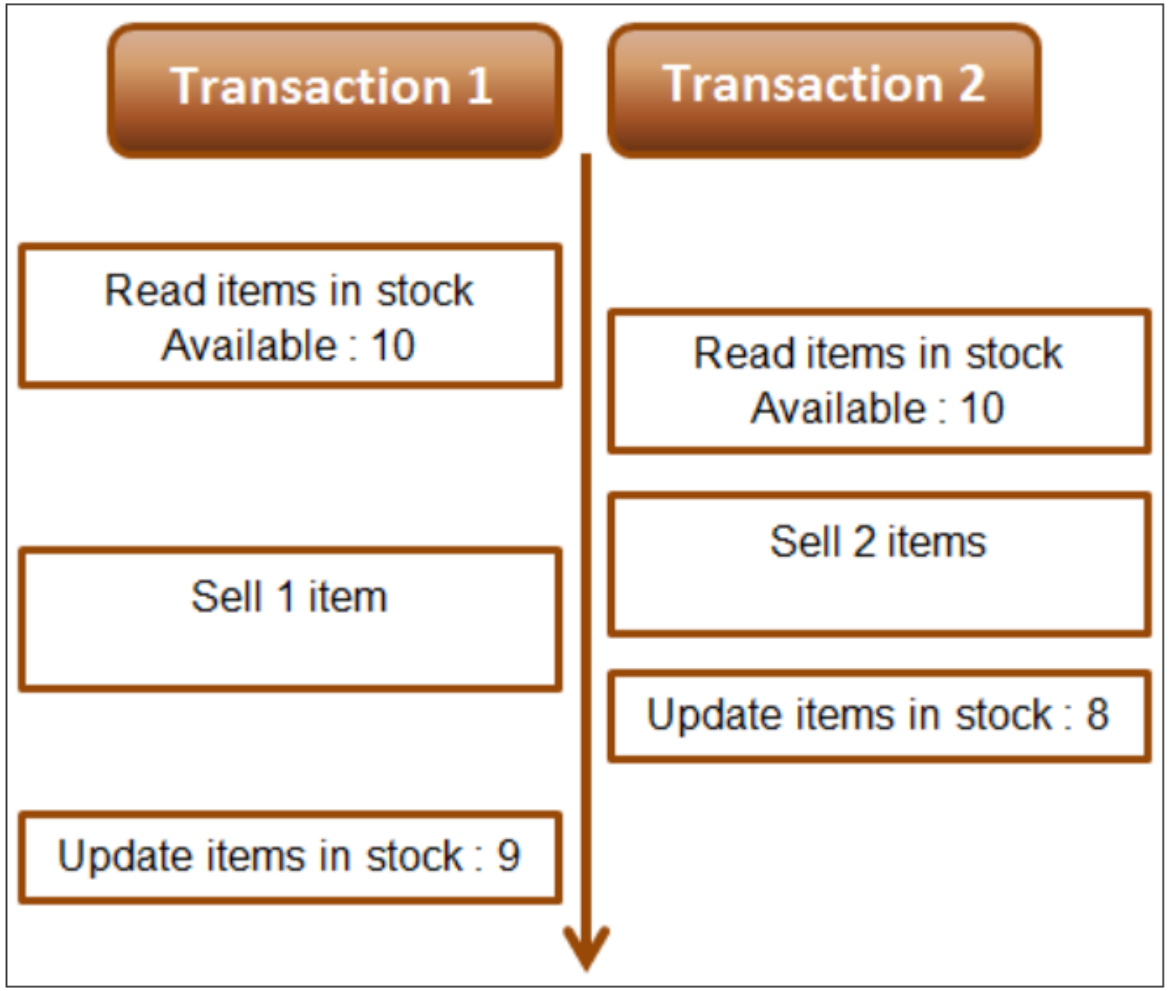
Definition

Ein Dirty Read („schmutziges Lesen“) tritt auf, wenn eine Transaktion Daten liest, die von einer anderen Transaktion geschrieben oder geändert wurden, jedoch noch nicht bestätigt (committed) sind.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

Lost Update



https://csharp-video-tutorials.blogspot.com/2015/08/sql-server-lost-update-problem.html

Lost Update

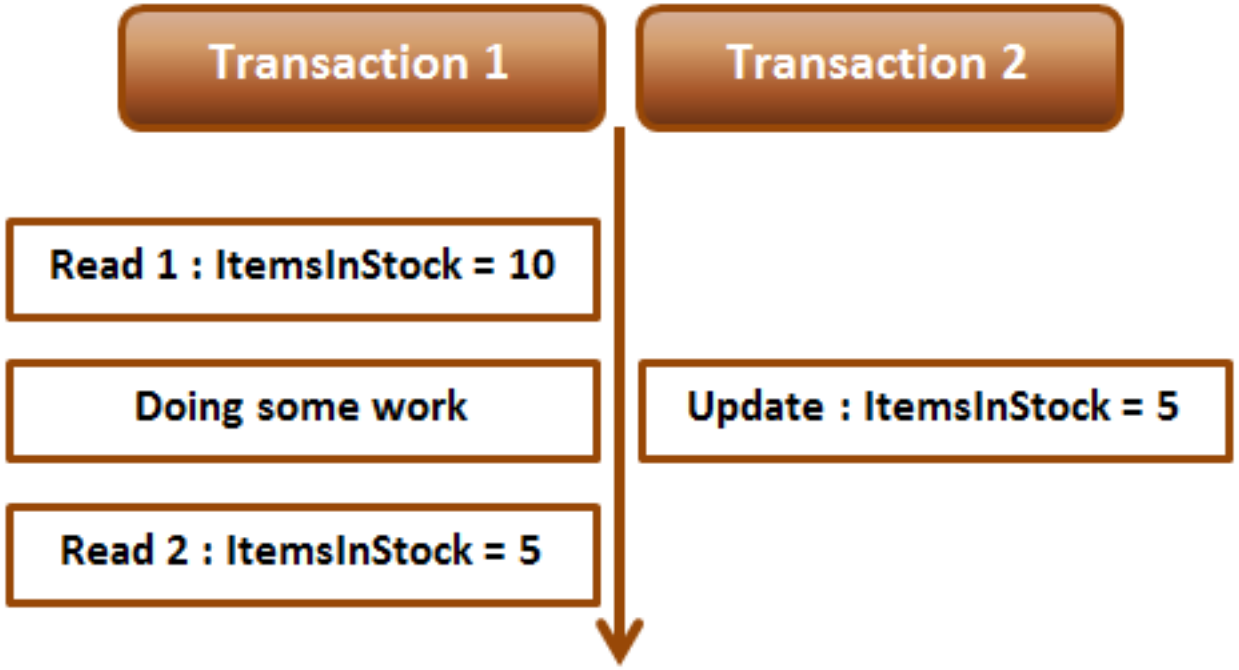
Definition

Ein Lost Update tritt auf, wenn zwei Transaktionen gleichzeitig dasselbe Objekt ändern und dabei eine Änderung verlorengeht, indem sie durch die zweite überschrieben wird.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

Non-Repeatable Read



Non-Repeatable Read

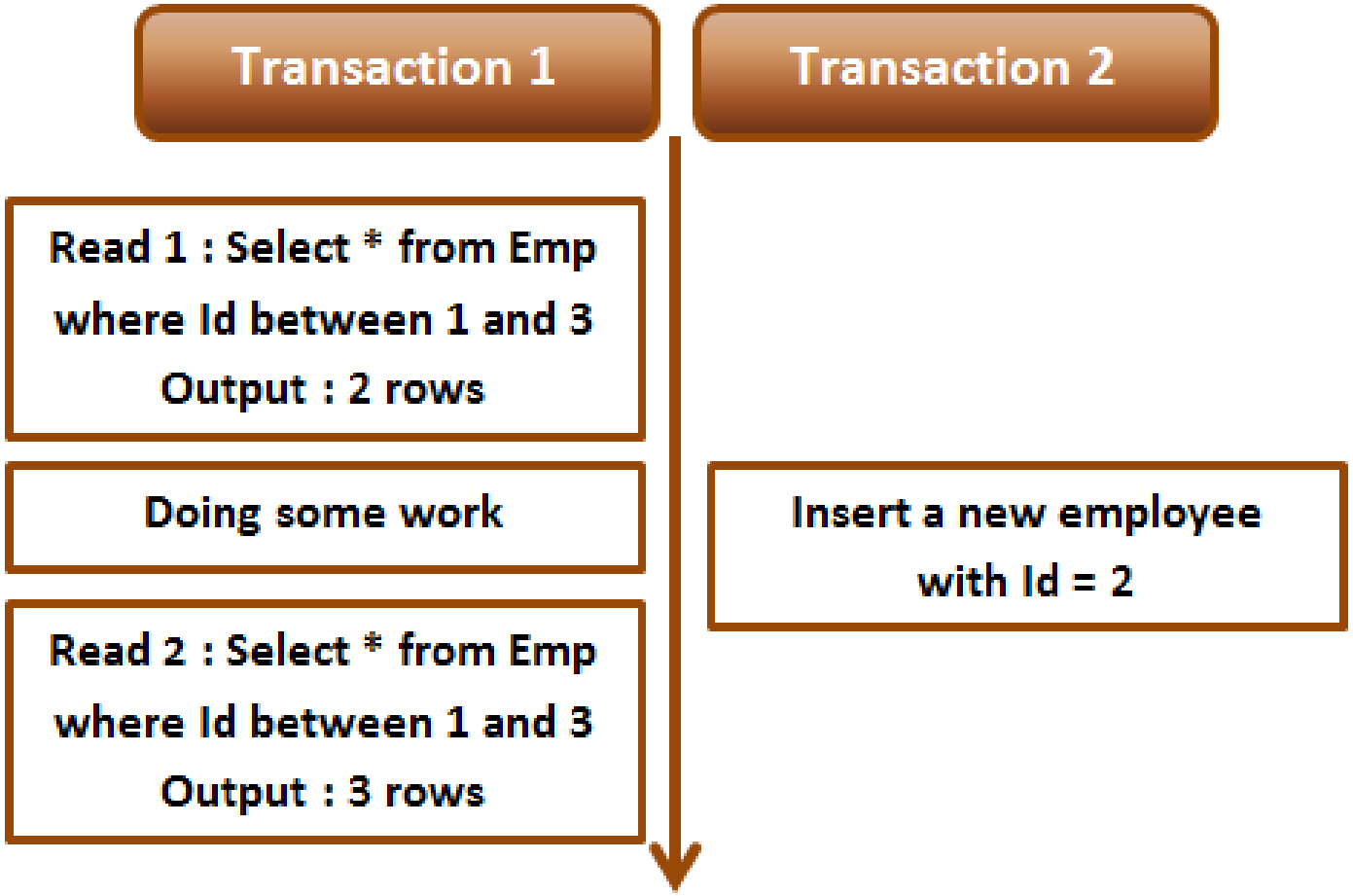
Definition

Als Non-Repeatable oder Unrepeatable Read wird ein Datenbankproblem bezeichnet, wenn eine Transaktion dieselbe Zeile in einer Tabelle zweimal liest und unterschiedliche Ergebnisse erhält.

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

Phantom Read



Phantom Read

Definition

Ein "Phantom Read" ist ein Datenbankproblem, das auftritt, wenn eine Transaktion dieselbe Abfrage zweimal ausführt und unterschiedliche Ergebnisse erhält, weil während der zweiten Ausführung eine andere Transaktion eine neue Zeile in der betreffenden Tabelle eingefügt hat.

Unterschied zu Non-Repeatable Read: Es kommt ein neuer Datensatz hinzu oder wird gelöscht. Beim unrepeatable Read wird nur ein Wert geändert.

Mehrbenutzerbetrieb führt bei unzureichender Abgrenzung (Isolation) zu Problemen:

- **Abhängigkeit von nicht festgeschriebenen Änderungen** z.B. von anderen nebenläufigen Transaktionen (*dirty read*)
- **Verlust** von Änderungen an den Daten (*lost update*)
- **Gelesene Daten zwischenzeitlich verändert** (*non-repeatable read & phantom read*)

Lösungsraum: Synchronisation / Concurrency Control

- Einfachste Lösung: Alle Transaktionen seriell ausführen

- Komplexere Lösung: Parallel Transaktionen nebenläufig ausführen

Herausforderung: Einen parallelen Ablaufplan (Schedule) finden, der den gleichen Effekt hat wie ein serieller Schedule.

Schedule (auch Historie)

Definition

- Ein Schedule ist ein Ablaufplan für eine oder mehrere Transaktionen
- Ein Schedule gibt die Abfolge der Datenbankoperationen an
- Serieller Schedule: Transaktionen laufen hintereinander ab
- Serialisierbarer Schedule: Schedule in dem Transaktionen verschränkt ausgeführt werden und der identische Effekt existiert wie bei einem seriellen Schedule

Schedules in Tabellen-Notation

Serieller Schedule

#	T_1	T_2
1	BOT	
2	Read(A)	
3	Write(A)	
4	Read(B)	
5	Write(B)	
6	commit	
7		BOT
8		Read(C)
9		Write(C)
10		Read(A)
11		Write(A)
12		commit

Paralleler Schedule

#	T_1	T_2
1	BOT	
2	Read(A)	
3		BOT
4		Read(C)
5	Write(A)	
6		Write(C)
7	Read(B)	
8	Write(B)	
9	Commit	
10		Read(A)
11		Write(A)
12		Commit

S_1: $r_1(A)$, $w_1(A)$, $r_1(B)$, $w_1(B)$, $r_2(C)$, $w_2(C)$, $r_2(A)$, $w_2(A)$

S_2 : $r_1(A)$, $r_2(C)$, $w_1(A)$, $w_2(C)$, $r_1(B)$, $w_1(B)$, $r_2(A)$, $w_2(A)$

Start A= 200; B=200; C=200

#	T_1	T_2	Bemerkung
1	BOT		
2	Read(A, a)		a=200
3	a:=a-50		a=150
4	Write(A, a)		A=150
5	Read(B, b)		b=200
6	b:= b+50		b=250
7	Write(B)		B=250
8	commit		
9		BOT	
10		Read(C,c)	c=200
11		c:=c+100	c=300
12		Write(C)	C=300
13		Read(A, a)	a=150
14		a:=a-100	a=50
15		Write(A)	A=50
16		commit	
17			

Ende A= 50; B=250; C=300

Start A= 200; B=200; C=200

#	T_1	T_2	Bemerkung
1	BOT		
2	Read(A)		a=200
3		BOT	
4		Read(C,c)	c=200
5	a:=a-50		a=150
6	Write(A)		A=150
7		c:=c+100	c=300
8		Write(C)	C=300
9	Read(B,b)		b=200
10	b:= b+50		b=250
11	Write(B)		B=250
12	Commit		
13		Read(A,a)	a=150
14		a:=a-100	a=50
15		Write(A)	A=50
16		Commit	

Ende A= 50; B=250; C=300

Identifikation von Konflikten : Welche Abhängigkeiten verursachen Probleme?

T_1	T_2
read A	
	read A

unabhängig von Reihenfolge

T_1	T_2
read A	
	write A

abhängig von Reihenfolge

T_1	T_2
	write A
read A	

abhängig von Reihenfolge

T_1	T_2
	write A
write A	

abhängig von Reihenfolge

- **Schreib-Lese-Abhängigkeit von $T_i \rightarrow T_j$**
  - Es existiert ein Objekt x, so dass in S $w_i(x)$ vor $r_j(x)$ kommt
  - Abkürzung: $wr_{i,j}(x)$

- **Lese-Schreib-Abhängigkeit Abhängigkeit von $T_i \rightarrow T_j$**
  - Es existiert ein Objekt x, so dass in S $r_i(x)$ vor $w_j(x)$ kommt
  - Abkürzung: $rw_{i,j}(x)$

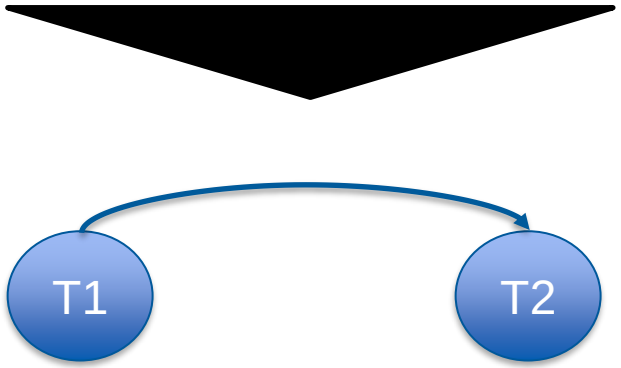
- **Schreib-Schreib-Abhängigkeit von $T_i \rightarrow T_j$**
  - Es existiert ein Objekt x, so dass in S $w_i(x)$ vor $w_j(x)$ kommt
  - Abkürzung: $ww_{i,j}(x)$

Objektivierbares Kriterium: Prüfung mittels eines Konfliktgraphen/Serialisierungsgraphen

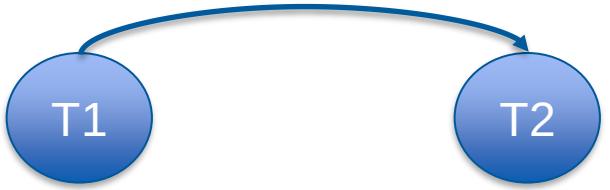
1. Schritt

$S_2 : r_1(A), r_2(C), w_1(A), w_2(C), r_1(B), w_1(B), r_2(A), w_2(A)$

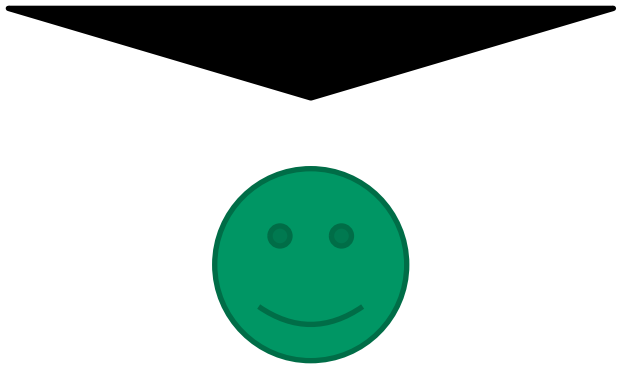
Überführen in einen Konfliktgraphen



2. Schritt



Ist Graph zyklenfrei?



Serialisierungsgraph / Konfliktgraph / Precedence Graph

Beim Erstellen eines Serialisierungsgraphen ziehen Sie Kanten zwischen den Knoten, die Transaktionen repräsentieren, basierend auf den folgenden Regeln:

Lesen-Schreiben-Konflikt (RW): Zeichnen Sie eine Kante von der Transaktion T1 zur Transaktion T2, wenn T1 ein Datenobjekt liest und T2 dieses Datenobjekt später schreibt, und zwar innerhalb ihrer jeweiligen Transaktionsausführungen

Schreiben-Lesen-Konflikt (WR): Zeichnen Sie eine Kante von der Transaktion T1 zur Transaktion T2, wenn T1 ein Datenobjekt schreibt und T2 dieses Datenobjekt später liest, und zwar innerhalb ihrer jeweiligen Transaktionsausführungen

Schreiben-Schreiben-Konflikt (WW): Zeichnen Sie eine Kante von der Transaktion T1 zur Transaktion T2, wenn T1 ein Datenobjekt schreibt und T2 dieses Datenobjekt später ebenfalls schreibt, und zwar innerhalb ihrer jeweiligen Transaktionsausführungen.

Anmerkung: Es wird immer vorwärtsgerichtet der Graph gebildet. Gehen Sie chronologisch vor!

#	T1	T2	T3
1	Read (A)		
2	Read (B)		
3		Read (A)	
4		Read (C)	
5	Write (B)		
6	Commit		
7			Read (B)
8			Read (C)
9			Write (B)
10			Commit
11		Write (A)	
12		Write (C)	
13		Commit	

Zeichnen Sie den Serialisierungsgraph. Ist der Schedule serialisierbar?

T_1	T_2	T_3	T_4
		read(b)	
		write(b)	
			write(b)
	read(b)		
read(a)			
read(c)			
write(a)			
write(c)			
		read(a)	
		write(c)	
	read(a)		
	write(c)		

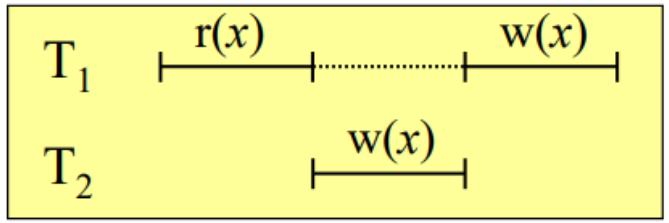
Konfliktserialisierbarkeit (conflict serializability)

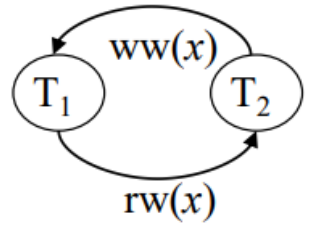
Die Serialisierbarkeit von Transaktionen ist ein Konzept in der Datenbanktechnologie, das sich auf die Fähigkeit bezieht, die Ausführung von Transaktionen so zu koordinieren, dass das Ergebnis dem Ergebnis einer seriellen Ausführung von Transaktionen entspricht.

Ein Schedule ist serialisierbar, falls der Serialisierungsgraph zyklenfrei ist. Man sagt in der DB-Sprache auch der Schedule ist „conflict serializable“.

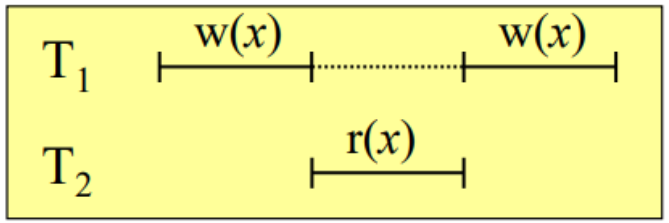
Beispiel für nicht-serialisierbare Schedules

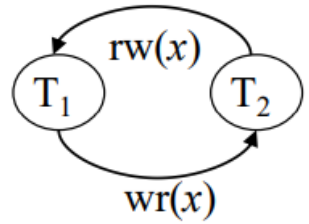
Lost Update: $S=(r_1(x), w_2(x), w_1(x))$



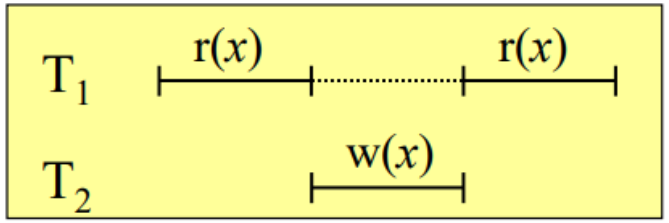


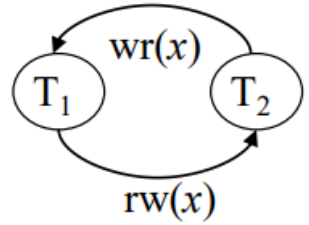
Dirty Read: $S=(w_1(x), r_2(x), w_1(x))$



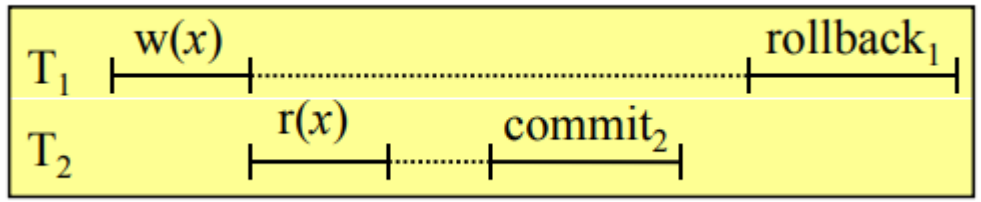


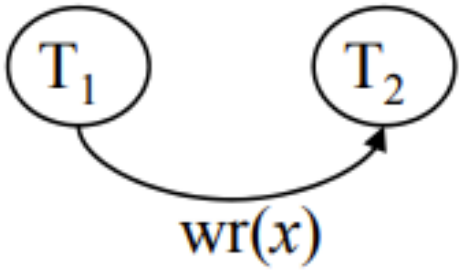
Non-repeatable Read: $S=(r_1(x), w_2(x), r_1(x))$





Weiteres Problem: Konfliktgraph alleine ist nicht ausreichend





Wo ist das Problem?

- ✓ Schedule ist serialisierbar
- ✓ Serialisierungsgraph ist zyklenfrei

Anforderung an rücksetzbare Schedules

Erkenntnis

Serialisierbarkeit alleine reicht nicht aus, wenn Transaktionen zurückgesetzt werden können!

Idee: Eine Transaktion T_i darf erst dann ihr COMMIT durchführen, wenn alle Transaktionen T_j, von denen sie Daten gelesen hat, beendet sind.

Problem der Idee

- Rücksetzbare Schedules können eine Lawine an Rollbacks in Gang setzen

- Verwaltungsaufwand für Serialisierungs-graphen ist zu hoch und rücksetzbare Schedules haben den Nachteil von langen Wartezeiten

Schritt	T_1	T_2	T_3	T_4	T_5
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_5(D)$	
8.					$r_5(D)$
9.	$abort_1$				

	T_1	T_2	Erläuterung
1	lockX(A)		
2	a_1 = read(A)		
3	write(A)		
4		lockS(A)	T_2 muss warten
5	lockX(B)		
6	read(B)		
7	unlockX(A)		
8		a_2 = read(A)	T_2 wecken
9		lockS(B)	T_2 muss warten
10	write(B)		
11	unlockX(B)		T_2 wecken
12		b_1 = read(B)	
13	commit		
14		unlockS(A)	
15		unlockS(B)	
16		commit	

RDBMSs verwenden Sperren, um sicherzustellen, dass verschachtelte Ausführung von Transaktionen äquivalent zu einer seriellen Ausführung ist

- **Lesesperre** (shared lock, read lock) **`S`** auf **`A`** erlaubt Transaktion T_i die Operation $r_i(A)$ auszuführen; mehrere Transaktionen können aufgrund einer Lesesperre lesend auf A zugreifen
- **Schreibsperre** (exclusive lock, write lock) **`X`** auf **`A`** erlaubt Transaktion T_i die Operation $w_i(A)$ auszuführen; nur eine Transaktion kann bei einer Schreibsperre lesend oder schreibend auf A zugreifen

Verträglichkeit von Sperren

Verträglichkeitstabelle

Es gelten folgende Verträglichkeiten zwischen Sperren:

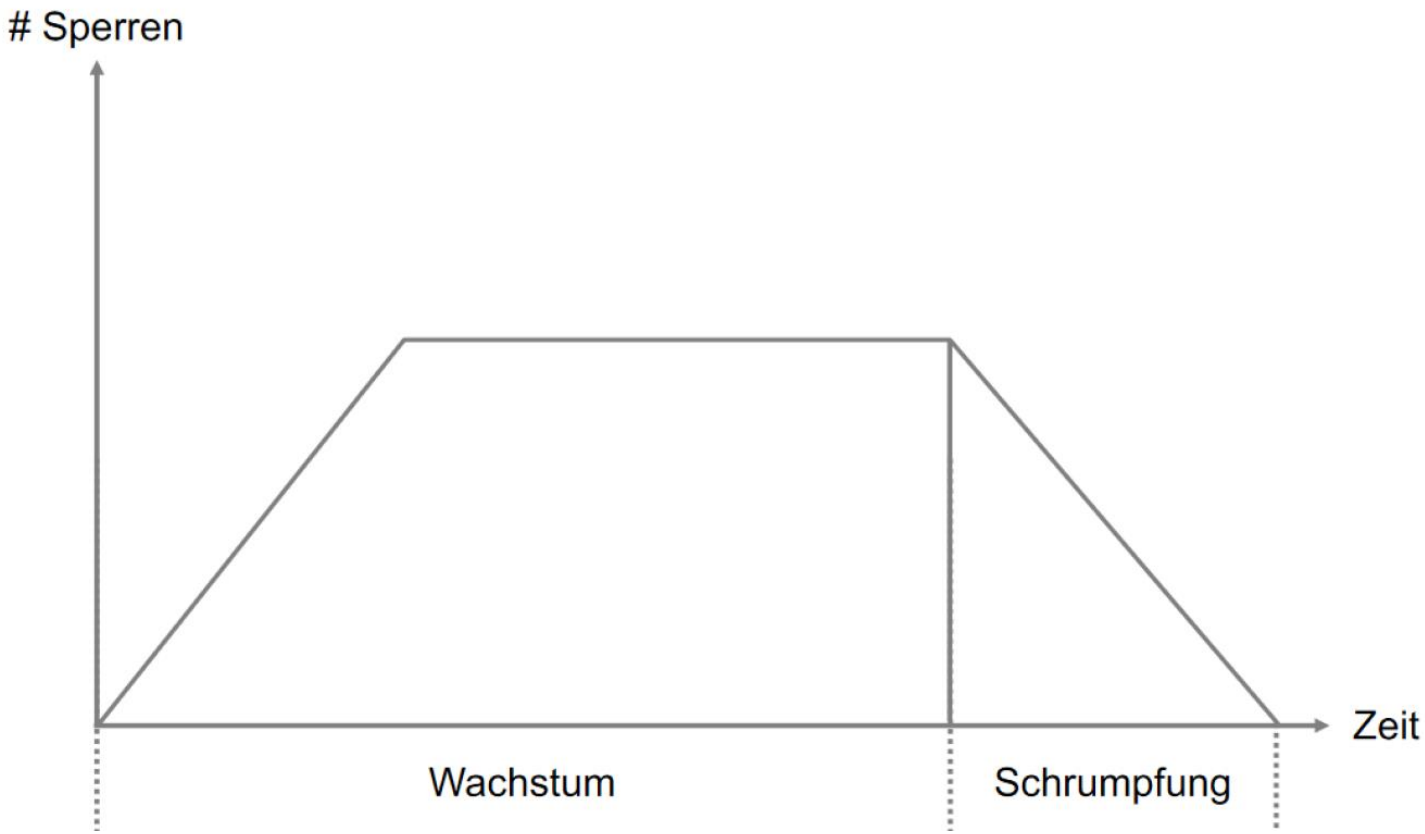
gesetzte Sperre	keine	S	X
Anforderung von S	ja	ja	nein
Anforderung von x	ja	nein	nein

- Besteht z.B. bereits eine Lesesperre auf einem Objekt, kann eine weitere Lesesperre, jedoch keine Schreibsperre angefordert werden
- Die anfordernde Transaktion muss dann warten, bis die bestehenden Lesesperren freigegeben werden

Zwei-Phasen-Sperrprotokoll (Two-phase Locking protocol)

- Jedes Objekt, das von einer Transaktion benutz werden soll, muss vorher entsprechend gesperrt werden
- Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an
- Eine Transaktion muss die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht – bis die Sperre gewährt werden kann
- Jede Transaktion durchläuft zwei Phasen:
    - Eine Wachstumsphase, in der sie Sperren anfordern, aber keine freigeben darf und
    - Eine Schrumpfphase, in der sie ihre bisher erworbene Sperren freigibt, aber keine weiteren Anfordern darf
- Bei EOT (Transaktionsende) muss eine Transaktion alle ihre Sperren zurückgeben

Wachstumsphase und **Schrumpfungsphase** im 2PL



Problem

	T_1	T_2	Erläuterung
1	lockX(A)		
2	a_1 = read(A)		
3	write(A)		
4		lockS(A)	T_2 muss warten
5	lockX(B)		
6	read(B)		
7	unlockX(A)		
8		a_2 = read(A)	T_2 wecken
9		lockS(B)	T_2 muss warten
	write(B)		
	unlockX(B)		T_2 wecken
12		b_1 = read(B)	
13			
14		unlockS(A)	
15		unlockS(B)	
16		commit	

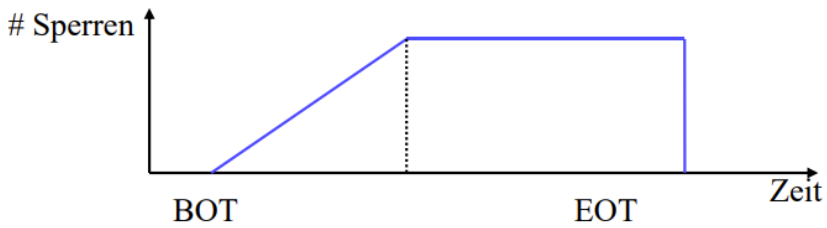
Rollback

- Transaktion 1 wird zurückgesetzt
- Transaktion 2 hat für A „schmutzig“ gelesen
- Beide Transaktionen müssten zurückgesetzt werden

Isolationslevel reich nicht aus!

Lösung: Strict 2 Phase Locking (S2PL) sowie Strong Strict 2 Phase Locking (SS2PL)

Striktes Zwei-Phasen-Sperrprotokoll



Alle Sperren werden bis zum Ende (Commit) der letzten Teiltransaktion gehalten.

- **Strict 2 Phase Locking:** Exclusive Locks werden bis zur letzten Teiltransaktion gehalten
- **Strong Strict 2 Phase Locking**: Shared und Exclusive Locks werden bis zur letzten Teiltransaktion gehalten

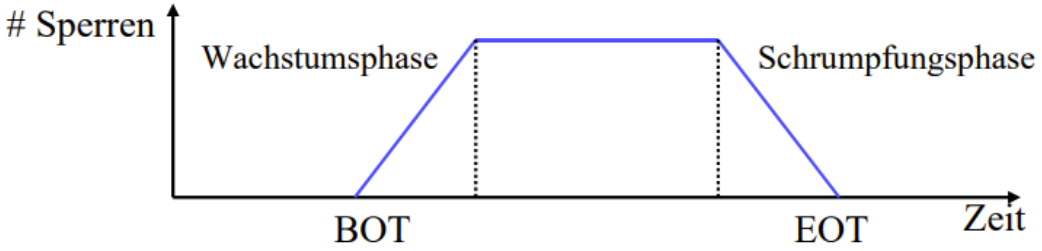
Verklemmungen (deadlocks)

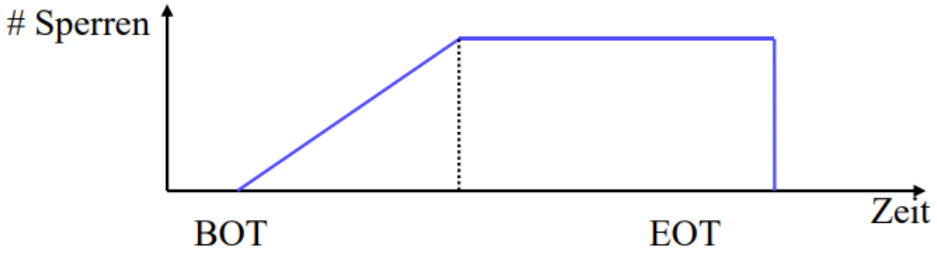
- Sperren können zu Verklemmungen (Deadlocks) führen, d.h. Transaktionen warten gegenseitig aufeinander

	T_1	T_2	Erläuterung
1	$\texttt{lockX}(A)$		
2		$\texttt{lockX}(B)$	
3		$b_2 = \texttt{read}(B)$	
4	$a_1 = \texttt{read}(A)$		
5	$\texttt{write}(A)$		
6	$\texttt{lockX}(B)$		T_1 wartet auf T_2
7		$\texttt{lockS}(A)$	T_2 wartet auf T_1

- RDBMS erkennt Verklemmungen (z.B. mittels Überwachung des Fortschritts) und löst sie auf, indem es eine der Transaktionen abbricht

Trade-Off: Performance vs. Strenge Isolierbarkeit





Isolationslevel ab SQL-92

Postgres

Isolation Level \ Phenomena	Dirty Read	Non-Repetable Read	Phantom Read	Lost Update
Read Uncommitted	Allowed But not in PG	YES	YES	YES
Read Committed	NO	YES	YES	YES
Repetable Read	NO	NO	NO	NO
Serializable	NO	NO	NO	NO

Isolationsebenen

```
BEGIN TRANSACTION [ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }];

COMMIT;
```

Default in Postgres:
Isolation Level Read Committed

Beispiel

```
-- start a transaction
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- deduct 1000 from account 1
UPDATE accounts
SET balance = balance - 1000
WHERE id = 1;

-- add 1000 to account 2
UPDATE accounts
SET balance = balance + 1000
WHERE id = 2;

-- commit the transaction
COMMIT;
```

PSQL: ROW EXCLUSIVE

ROW EXCLUSIVE ist eine Sperrstufe in PostgreSQL, die verhindert, dass andere Transaktionen die gesperrte Zeile aktualisieren oder löschen können, aber es erlaubt anderen Transaktionen, Lesezugriff auf die Zeile zu haben. Hier ist ein Beispiel für die Verwendung von ROW EXCLUSIVE in PostgreSQL.

Nicht im SQL-Standard definiert!

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Hands-on Beispiel

Update

```
BEGIN TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
UPDATE transaktion.accounts
SET balance = 400
WHERE id = 1;



COMMIT;
```

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE transaktion.accounts
SET balance = 600
WHERE id = 1;
COMMIT;
```

Update

```
BEGIN TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
UPDATE transaktion.accounts
SET balance = 400
WHERE id = 1;



COMMIT;
```

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET LOCAL lock_timeout = '4s';
UPDATE transaktion.accounts
SET balance = ((select balance from transaktion.accounts WHERE
id = 1) -50)
WHERE id = 1;

COMMIT;
```