Handel, Banken, Ticketsystem (Teil 2)

VL Big Data Engineering
(vormals Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

23. Juni 2022

Spickzettel für Teil 1

1. Transaktion: bündelt mehrere SQL-Statements
2. ACID: Atomarität, Konsistenz, Isolation, Dauerhaftigkeit. Für jede Transaktion müssen die ACID-Eigenschaften garantiert werden.
3. Aber wie machen wir das? Insbesondere: wie garantieren wir **automatisch** Isolation?
4. Ausführungsplan (Historie): konkrete Verschränkung von Lese-/Schreiboperationen
5. Serieller Ausführungsplan: Transaktionen werden nacheinander ausgeführt
6. Konfliktserialisierbarer Ausführungsplan: äquivalent zu einem seriellen Ausführungsplan
7. Konflikt-Graph: aggregierte Form eines Ausführungsplan: Knoten entsprechen Transaktionen
8. Kein Zyklus im Konflikt-Graph $\Rightarrow$ zugehöriger Ausführungsplan ist konfliktserialisierbar

Mögliche Isolationsprobleme und ihre Lösungen

Probleme:

1. Dirty Read
2. Non-Repeatable Read
3. Cascading Rollback

Lösung: tupelbasiertes Sperren (Locking)

aber wie genau? kurzzeitig, langzeitig, 2PL, S2PL?

Durch diese Lösung entstehen **zwei neue Probleme**:

4. Phantom Problem

Lösung: prädikatbasierte Sperren (Locking)

5. Deadlock Problem

Lösung: Wartegraph


phantom problem
deadlocks
dirty read

Dirty Read (Schmutziges Lesen)

Dirty Read (Schmutziges Lesen)

Mit **Dirty Read** bezeichnen wir das Lesen eines Wertes durch eine Transaktion, der von einer anderen nicht committeten oder abgebrochenen Transaktion geschrieben wurde. D.h. es wurde ein Wert gelesen, der im Sinne der Isolation noch nicht für andere Transaktionen hätte sichtbar sein dürfen.

Beispiel:

AP: $w_1(A) \rightarrow \underbrace{r_2(A)}_{\text{dirty read}} \rightarrow w_2(B) \rightarrow \underbrace{r_2(B)}_{\textbf{kein dirty read}} \rightarrow \underbrace{r_1(B)}_{\text{dirty read}} \rightarrow c_2 \rightarrow c_1$

Wie vermeiden wir dirty reads?

Non-Repeatable Read (Nichtwiederholbares Lesen)

Non-Repeatable Read (Nichtwiederholbares Lesen)

Mit **Non-Repeatable Read** bezeichnen wir das Phänomen, dass eine Transaktion wiederholt dasselbe Datenobjekt aus der Datenbank liest, aber möglicherweise unterschiedliche Werte zurückbekommt. D.h. die Transaktion sieht beim wiederholten Lesen einen veränderten Wert, was im Sinne der Isolation nicht möglich sein sollte, da Änderungen nebenläufig ausgeführter Transaktionen (inklusive committeter Transaktionen) nicht sichtbar sein sollten.

Beispiel:

AP: $r_1(A) \rightarrow w_2(B) \rightarrow \underbrace{r_1(A)}_{\text{repeatable read}} \rightarrow w_2(A) \rightarrow c \rightarrow \underbrace{r_1(A)}_{\text{non-repeatable read}} \rightarrow \ldots$

Wie vermeiden wir non-repeatable reads?

Cascading rollback (Kaskadierendes Zurücksetzen)

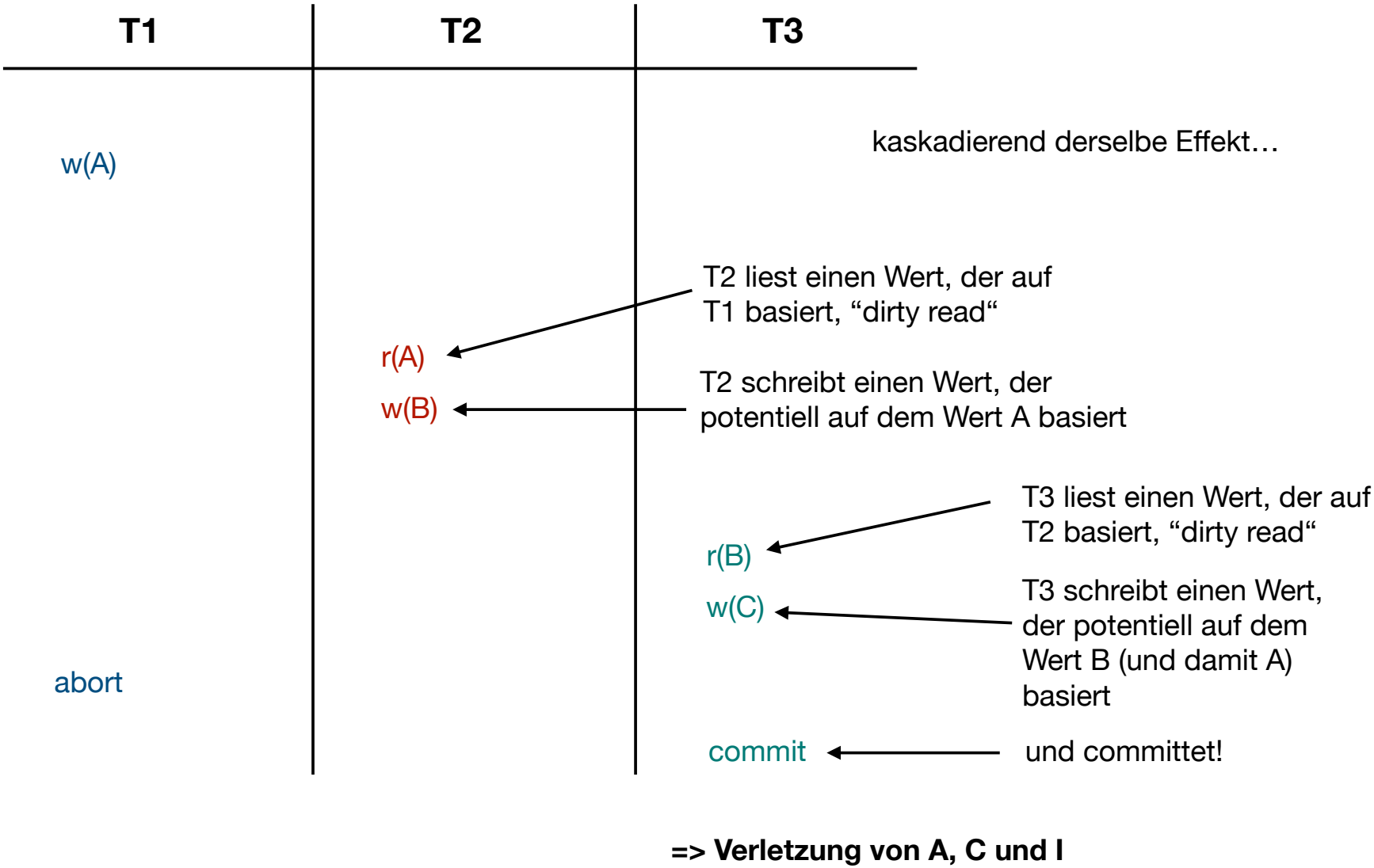
Cascading rollback (Kaskadierendes Zurücksetzen)

Mit **Cascading rollback** bezeichnen wir das Phänomen, dass eine Transaktion Dirty Reads durchführt und darauf basierend neue Werte schreibt. D.h. die Transaktion führt basierend auf nicht-committeten Werten neue Schreiboperationen durch. Dieses Problem kann dazu führen, dass eine ganze Kaskade von Transaktionen zurückgesetzt werden muss.

Beispiel:

AP: $w_1(A) \rightarrow \underbrace{r_2(A)}_{\text{dirty read}} \rightarrow \underbrace{w_2(B)}_{\text{potentiell ungültiger Wert}} \rightarrow a$

Erweitertes Beispiel für Cascading Rollback



=> Verletzung von A, C und I

Wie vermeiden wir cascading rollbacks?

Tupelbasiertes Sperren (Tuple-based Locking)

Tupelbasiertes Sperren (Tuple-based Locking)

1. Jede Operation, die ein Tupel lesen will, muss zunächst eine **Lesesperre (shared lock, R oder S)** für dieses Tupel bekommen.
2. Jede Operation, die ein Tupel ändern will, muss zunächst eine **Schreibsperre (exclusive lock, X)** für dieses Tupel bekommen.
3. Erhält eine Operation eine Lese- oder Schreibsperre nicht, muss diese Operation solange warten, bis sie die Sperre zugeteilt bekommt.
4. Alle Sperren müssen spätestens bei Transaktionsende zurückgegeben werden.

Notation: Eine Lesesperre von Transaktion T_i auf Datenobjekt A_j notieren wir durch $getRLock_i(A_j)$ oder $getSLock_i(A_j)$. Eine Schreibsperre von Transaktion T_i auf Datenobjekt A_j notieren wir durch $getXLock_i(A_j)$. Eine Sperre wird mit der Operation $releaseLock_i(A_j)$ zurückgegeben.

Kompatibilität von Sperren

Kompatibilität von Lese-/Schreibsperren

Für jedes Tupel dürfen zu jedem Zeitpunkt:

1. falls keine Schreibsperre existiert: beliebige viele Lesesperren zugelassen werden,
2. falls keine Lese- oder Schreibsperre existiert: eine Schreibsperre zugelassen werden.

Beim Anfordern von Sperren muss folgende Kompatibilitätsmatrix beachtet werden:

		angeforderte Sperre	
		S	X
existierende Sperre	S	✓	✗
	X	✗	✗
	keine	✓	✓

✗ : Sperre wird nicht erteilt => Transaktion muss warten

✓ : Sperre wird erteilt => Transaktion darf weiterrechnen

Kurzzeitige Sperren

Kurzzeitige Sperren

Bei einer Leseoperation $r_i^k(A_j)$ wird von T_i direkt davor in Operation $k-1$ eine Sperre auf das Tupel A_j angefordert und direkt nach der Leseoperation $(k+1)$ zurückgegeben. (ebenso für Schreiboperationen)

Beispiele:

```
getSLock(A)
read(A)
releaseLock(A)
```

oder:

```
getXLock(A)
write(A, 100)
releaseLock(A)
```

Langzeitige Sperren: Two-Phase-Locking (2PL)

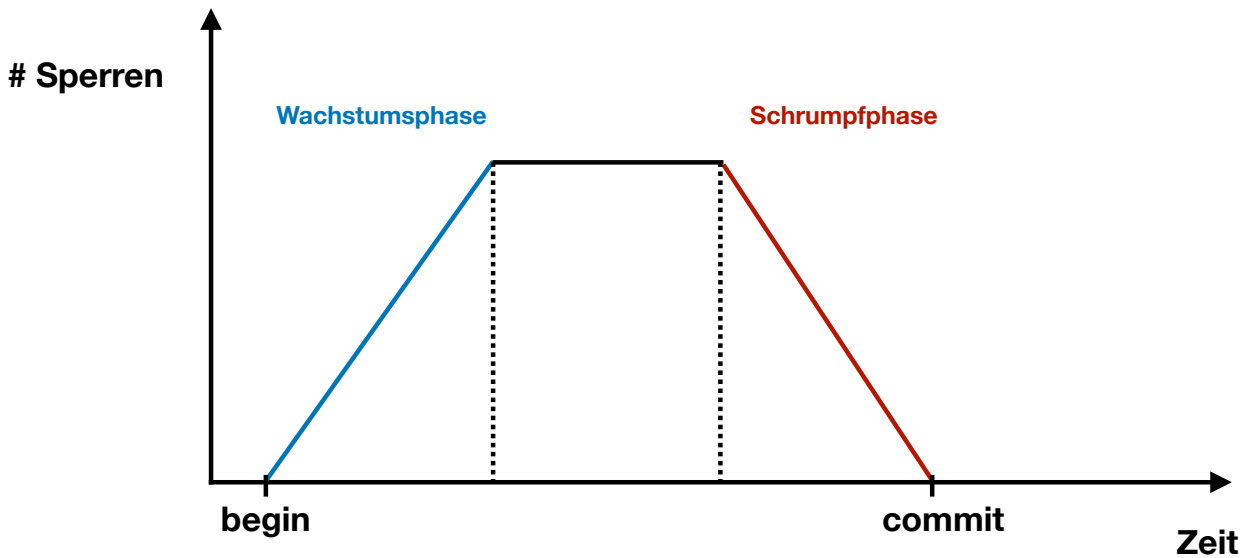
Two-Phase-Locking (2PL)

Die Laufzeit einer Transaktion teilt sich in zwei zeitliche Phasen auf:

1. **Wachstumsphase:** Die Transaktion kann Sperren anfordern.
2. **Schrumpfphase:** Die Transaktion kann Sperren zurückgeben.

Sobald eine Transaktion die erste Sperre zurückgibt, beginnt die Schrumpfphase.

Zeitliche Übersicht für eine Transaktion unter 2PL:



Einschub: 2PL vs Konfliktserialisierbarkeit

2PL vs Konfliktserialisierbarkeit

2PL lässt nur Ausführungspläne zu, die konfliktserialisierbar sind.

OK, dann ist alles gut, oder?

Leider nein, denn ein Ausführungsplan kann auch abgebrochene Transaktionen enthalten! Mit anderen Worten: jede Transaktion kann jederzeit vom Nutzer/Anwender/Datenbanksystem abgebrochen werden.

Konfliktserialisierbarkeit heißt nur, dass der Ausführungsplan äquivalent zu einem seriellen Ausführungsplan ist, aber nicht, dass die in diesem Ausführungsplan abgebrochenen Transaktionen ihre eingebrachten Änderungen wieder zurücknehmen!

Dieses Problem löst 2PL nicht!

Striktes Two-Phase-Locking (Strict 2PL, S2PL)

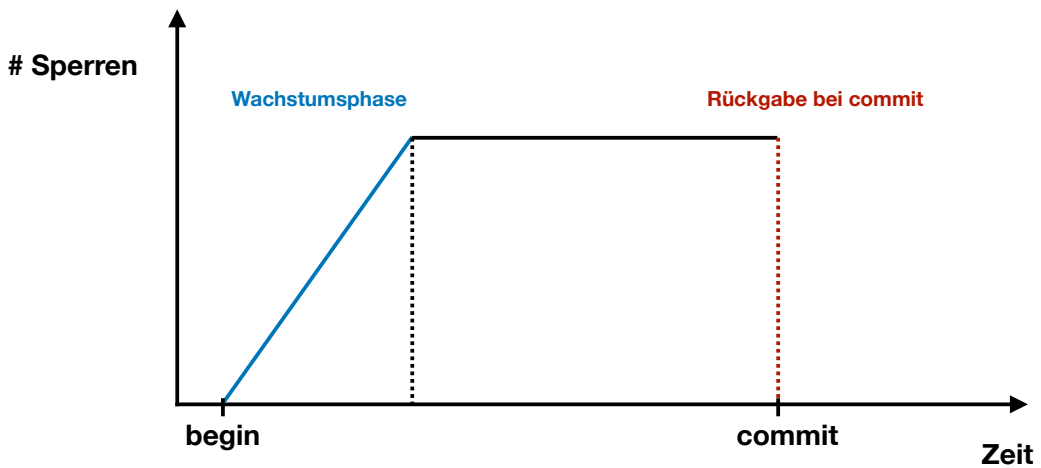
Striktes Two-Phase-Locking (Strict 2PL, S2PL)

Die Laufzeit einer Transaktion hat nur eine zeitliche Phase:

1. **Wachstumsphase:** Für jede Operation $o_i^k(A_j)$ muss T_i spätestens zum Zeitpunkt $k-1$ die entsprechende Sperre auf A_j anfordern.

Alle Sperren werden erst bei Ende der Transaktion (durch commit oder abort) zurückgegeben.

Zeitliche Übersicht für eine Transaktion unter S2PL:



S2PL verhindert Ausführungspläne mit kaskadierendem Zurücksetzen.

Mögliche Isolationsprobleme und ihre Lösungen

Probleme:

1. Dirty Read
2. Non-Repeatable Read
3. Cascading Rollback

Lösung: tupelbasiertes Sperren (Locking)

aber wie genau? kurzzeitig, langzeitig, 2PL, S2PL?

Durch diese Lösung entstehen **zwei neue Probleme**:

4. Phantom Problem

Lösung: prädikatbasierte Sperren (Locking)

5. Deadlock Problem

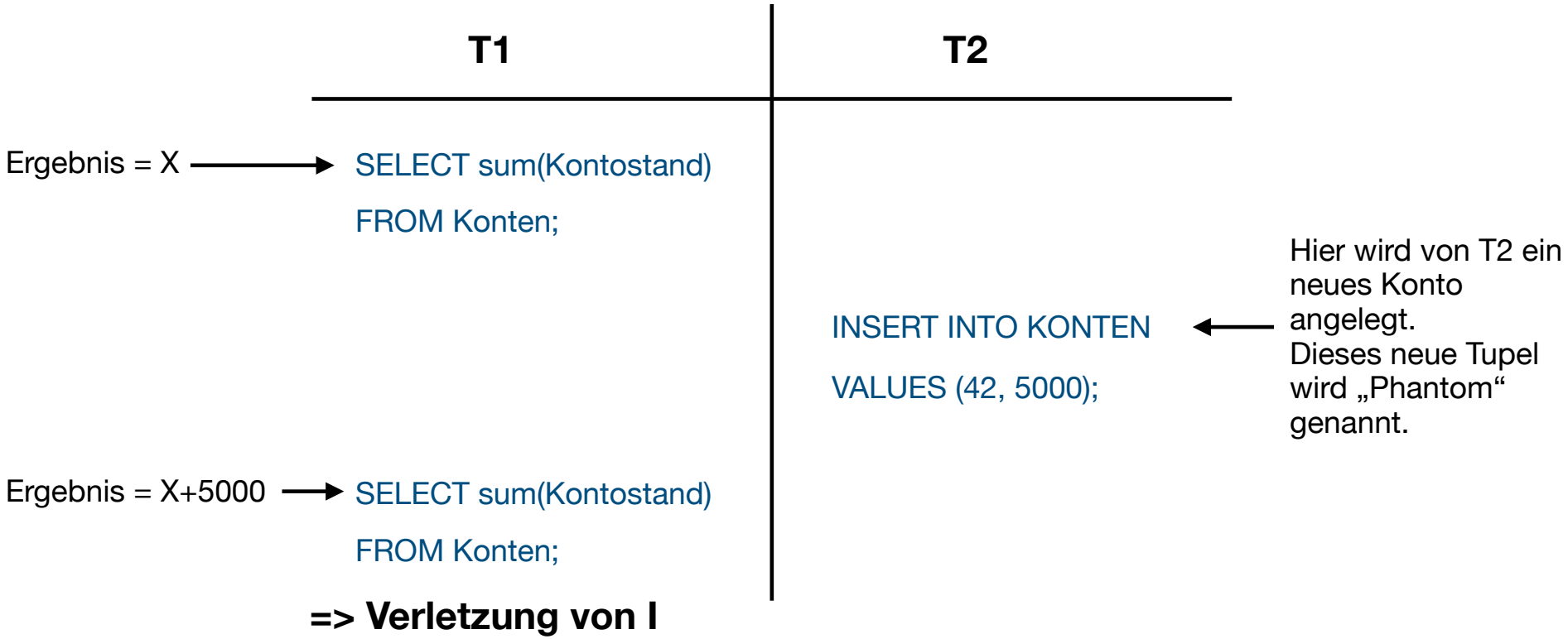
Lösung: Wartegraph

Phantom problem (Phantomproblem)

Phantom problem (Phantomproblem)

Mit **Phantomproblem** bezeichnen wir das Phänomen, dass eine Transaktion mehrfach eine Menge von Tupeln liest und potenziell unterschiedliche Ergebnisse erhält. Dieses Problem ist ähnlich zu Non-Repeatable Read, allerdings auf Tabellenebene.

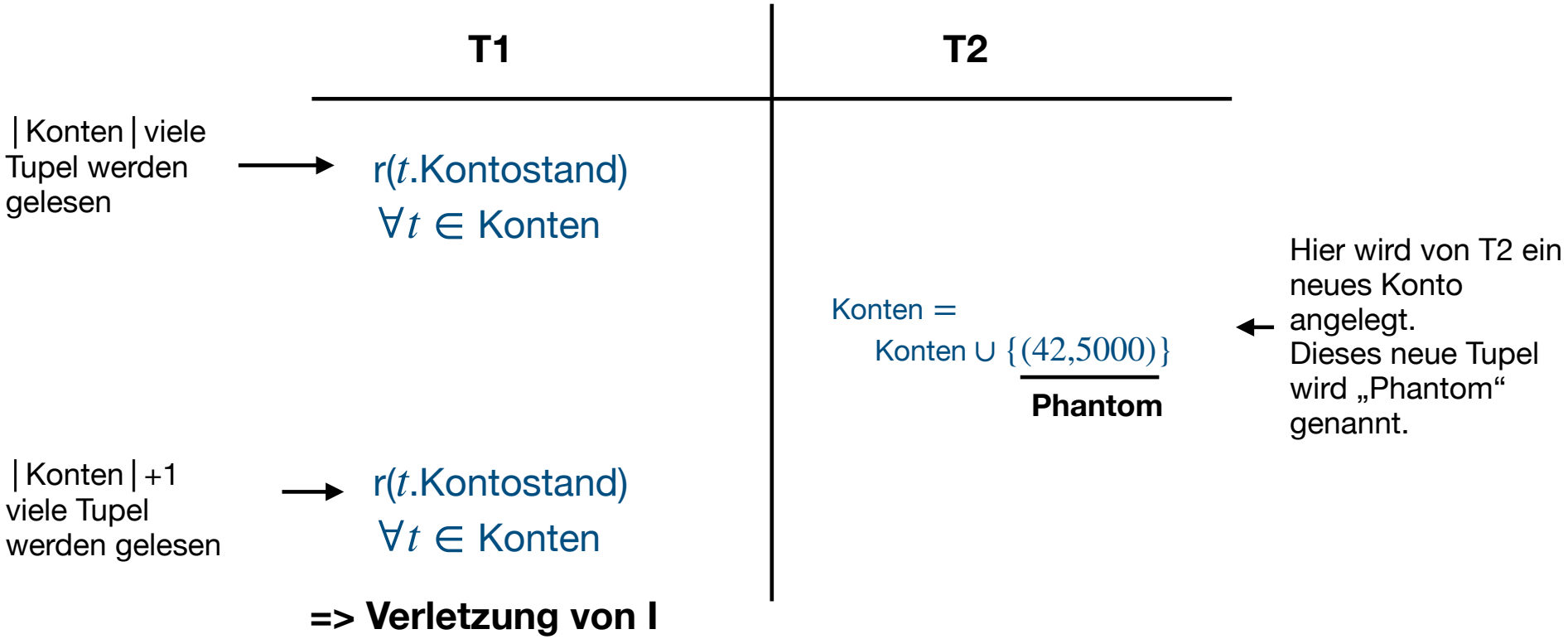
Beispiel: Phantomproblem



Was ist der Unterschied, zu den Szenarios, die wir bisher betrachtet haben?

Wir betrachten hier nicht mehr nur Lese-/Schreiboperationen, die **ein konkretes, garantiert existierendes Tupel** lesen oder schreiben, sondern Situationen, wo über die jeweilige WHERE-Klausel **sich die Menge der Tupel potentiell ändert**.

Sperren mehrerer Tupel

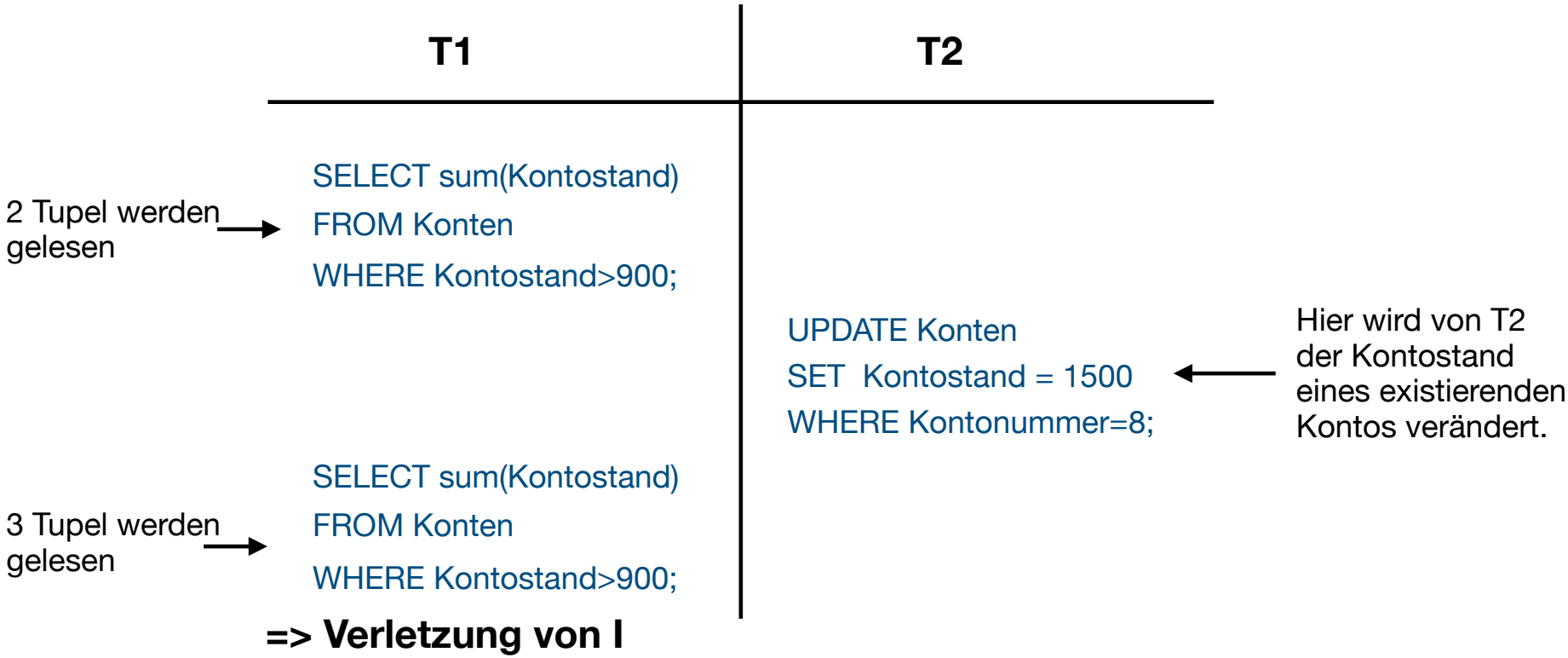


Was bedeutet dies für das Sperren von Tupeln?

Ein einzelnes Statement in einer Transaktion fordert u.U. mehr als eine Lese- und/oder Schreibsperre an.
In dem Beispiel bedeutet $\forall t \in$ Konten r(t.Kontostand), dass für jedes zu diesem Zeitpunkt in der Relation Konten existierende Tupel **konzeptuell** eine separate Leseoperation ausgeführt wird.

Phantomproblem ohne Insert oder Delete

Beispiel: Dieser Ausführungsplan wird mit S2PL zugelassen!



Das Problem ist hier, dass innerhalb der WHERE-Klausel unterschiedliche Tupel selektiert werden. Das Prädikat Kontostand>900 führt je nach Ausführungszeitpunkt zu unterschiedlichen Ergebnissen. Somit wird Isolation verletzt.

Prädikatsperren (Predicate Locking)

Prädikatsperren (Predicate Locking)

Um zu verhindern, dass eine Transaktion mit SQL-Statements, die auf mehrere Tupel zugreifen, ein Phantom sieht, müssen zusätzlich für solche Statements die entsprechenden Wertebereiche gesperrt werden. Dies können wir durch Prädikate ausdrücken.

Beispiele:

```
SELECT   sum(Kontostand)
FROM     Konten;
```

bedeutet eigentlich:

```
SELECT   sum(Kontostand)
FROM     Konten
WHERE    True;
```

⇒ Vor dem Lesen die ganze Relation Konten sperren!

```
SELECT   sum(Kontostand)
FROM     Konten
WHERE    Kontostand>900;
```

⇒ Vor dem Lesen den Bereich sperren, für den gilt, dass der Kontostand größer als 900 ist!

Kompatibilität von Prädikatsperren

Notation: Das Anfordern einer Prädikatsperre zum Lesen von Transaktion T_i auf Prädikat p wird als $getPSLock_i(p)$ notiert. $getPXLock_i(p)$ fordert die analoge Prädikatsperre zum Schreiben an.

Kompatibilität von Prädikatsperren

1. Eine neue Prädikatlesesperre auf Relation R wird zugelassen, wenn es keine aktive Prädikatschreibsperre einer anderen Transaktion mit Prädikat q gibt für die $\{r \in D \mid p(r) = true\} \cap \{r \in D \mid q(r) = true\} \neq \emptyset$ gilt. $D = D_1 \times \ldots \times D_n$
2. Eine neue Prädikatschreibsperre auf Relation R wird zugelassen, wenn es keine aktive Prädikatsperre (lesen oder schreiben) einer anderen Transaktion mit Prädikat q gibt für die $\{r \in D \mid p(r) = true\} \cap \{r \in D \mid q(r) = true\} \neq \emptyset$ gilt.

Beispiel:
$getPSLock_1(R.x = 0)$ und $getPXLock_2(R.y = 42)$ sind nie miteinander kompatibel, unabhängig davon, ob das Tupel $(0, 42)$ in der Relation enthalten ist.

Mögliche Isolationsprobleme und ihre Lösungen

Probleme:

1. Dirty Read
2. Non-Repeatable Read
3. Cascading Rollback

Lösung: tupelbasiertes Sperren (Locking)

aber wie genau? kurzzeitig, langzeitig, 2PL, S2PL?

Durch diese Lösung entstehen **zwei neue Probleme**:

4. Phantom Problem

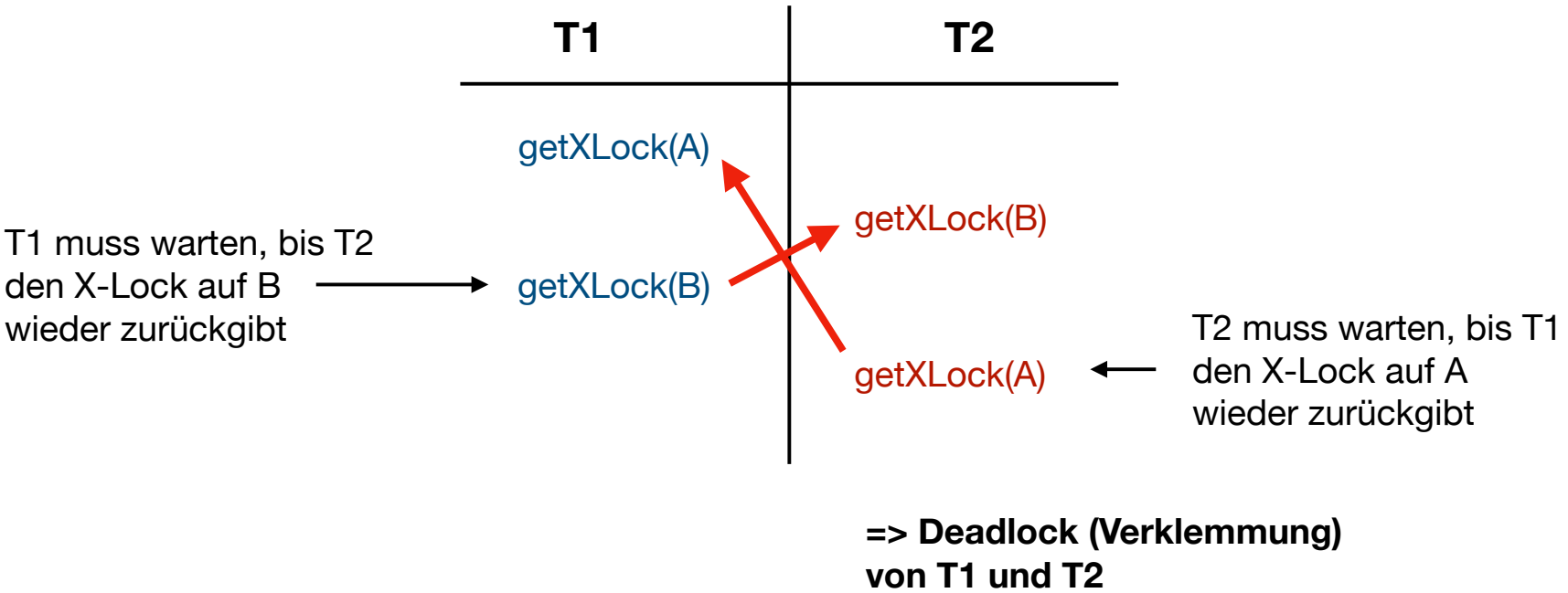
Lösung: prädikatbasierte Sperren (Locking)

5. Deadlock Problem

Lösung: Wartegraph

Verklemmung (Deadlock)

Beispiel: Dieser Ausführungsplan wird mit S2PL zugelassen!



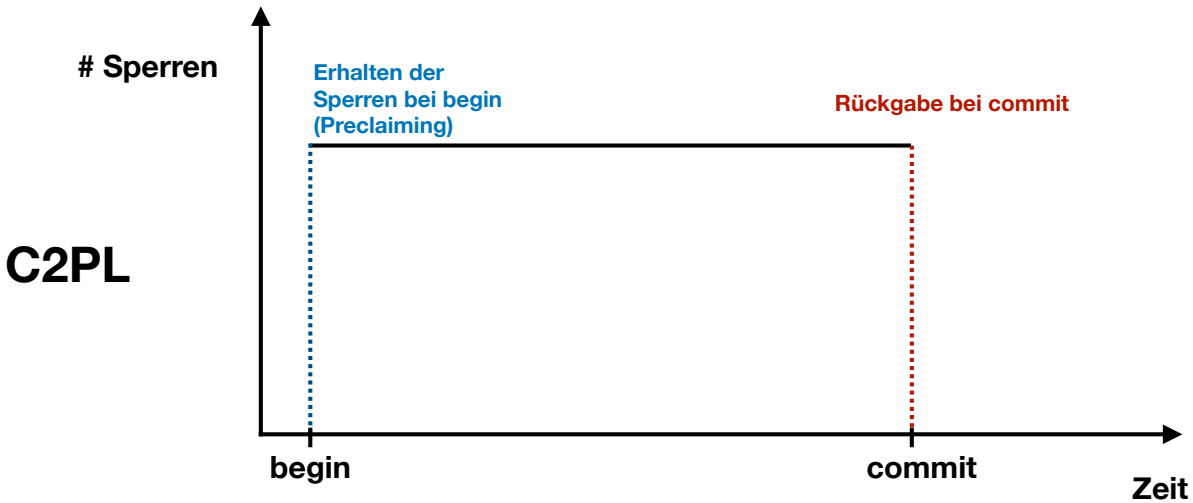
Da T1 und T2 gegenseitig auf sich warten, werden sie niemals weiterrechnen.

Konservatives Two-Phase-Locking (Conservative 2PL, C2PL, preclaiming)

Konservatives Two-Phase-Locking

Alle Sperren werden bei begin angefordert. Falls die Transaktion nicht alle Sperren bekommt, rechnet sie nicht los.
Sperren werden nicht vor dem commit zurückgegeben.

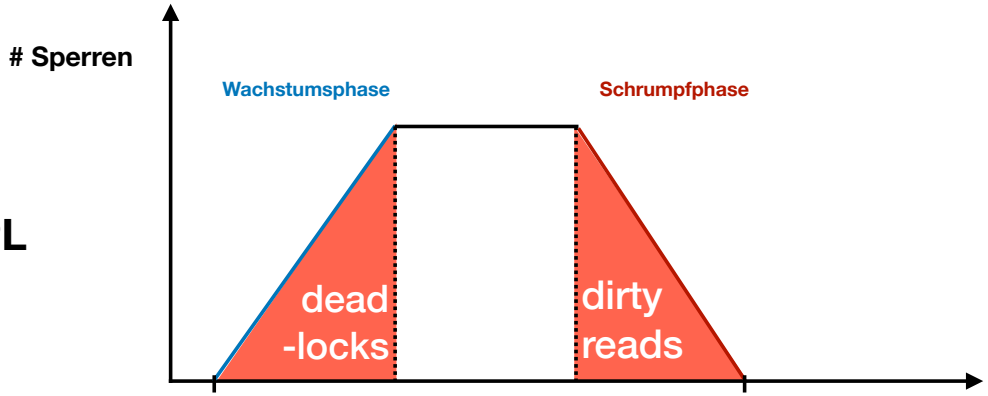
Zeitliche Übersicht für eine Transaktion unter C2PL:



C2PL verhindert Deadlocks.

Übersicht über die verschiedenen 2PL-Varianten

2PL

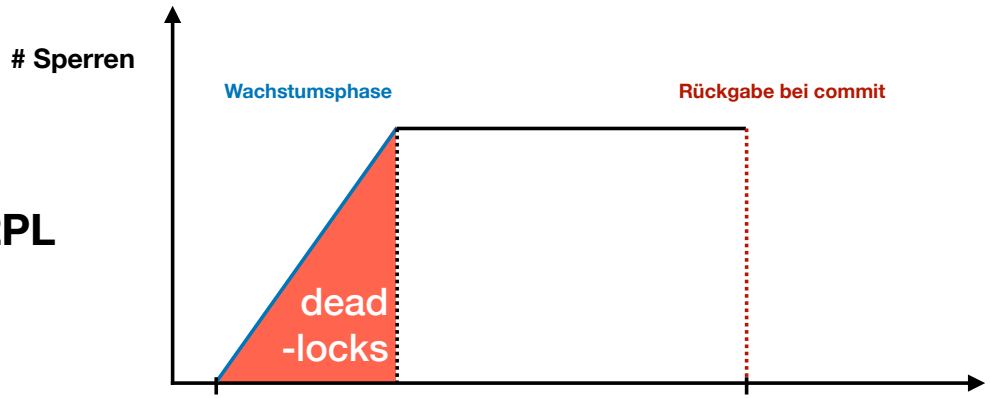


deadlocks sind möglich

dirty reads durch andere Transaktionen sind möglich

Sperren werden bei Bedarf angefordert und früh zurückgegeben

S2PL

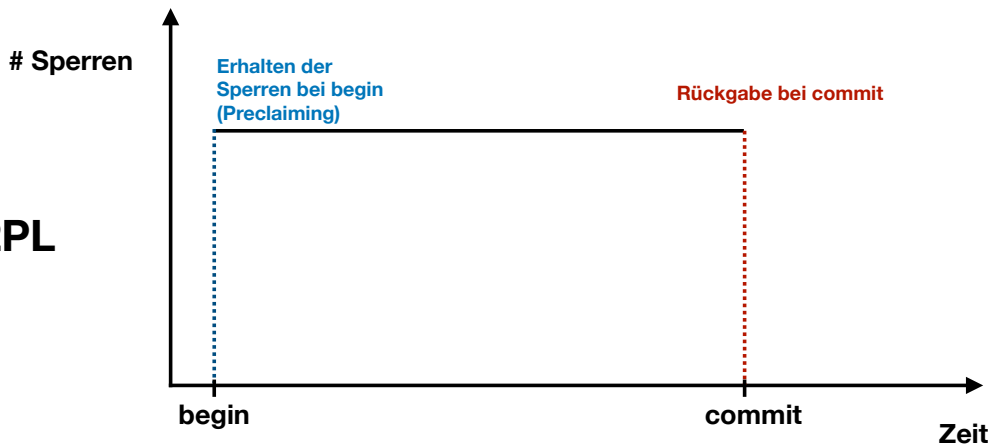


deadlocks sind möglich

dirty reads durch andere Transaktionen sind **nicht** möglich

Sperren werden bei Bedarf angefordert und bei commit zurückgegeben

C2PL



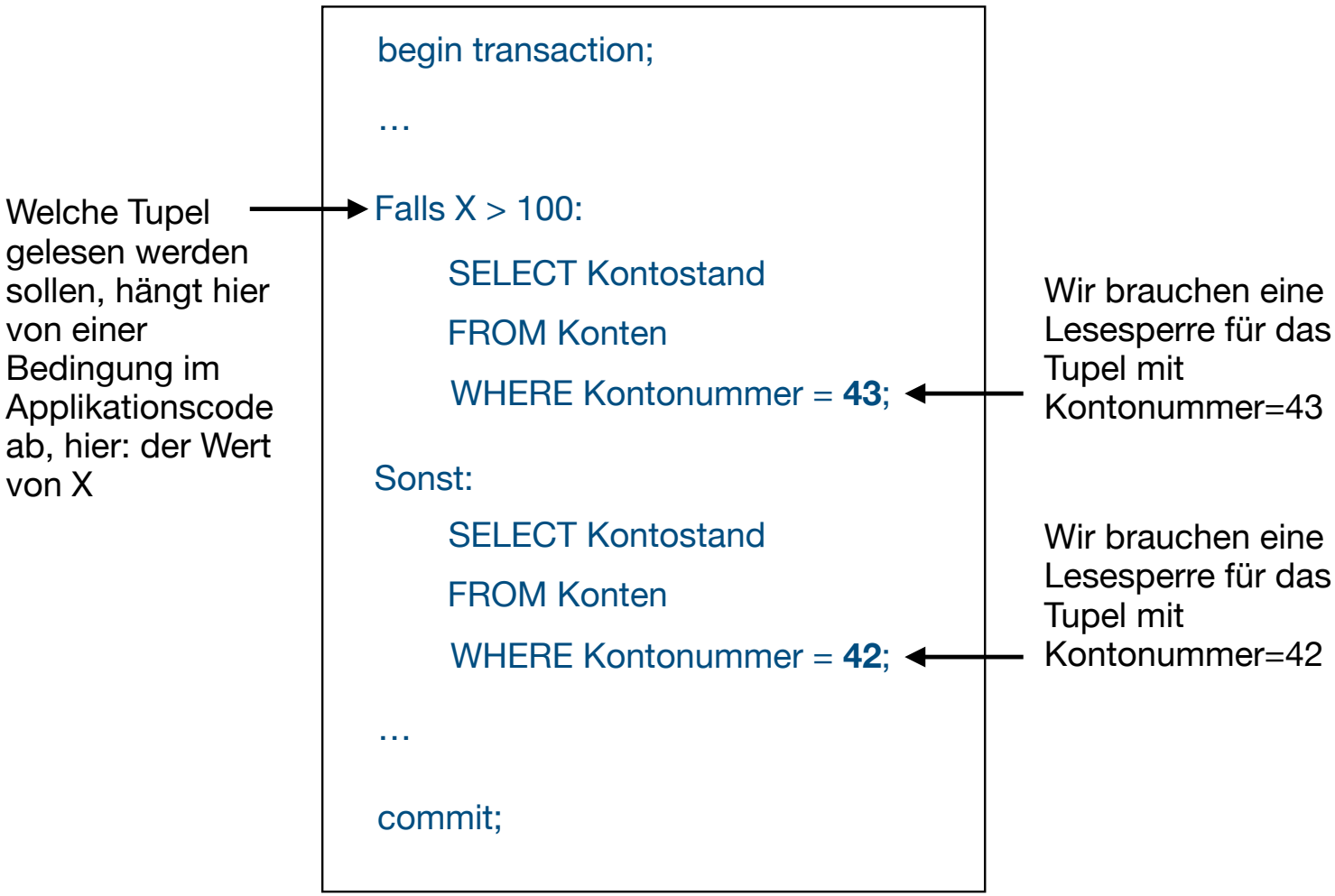
deadlocks sind **nicht** möglich

dirty reads durch andere Transaktionen sind **nicht** möglich

Sperren werden bei begin angefordert und bei commit zurückgegeben

Beispiel: Warum Preclaiming eine schlechte Idee ist

Beispielcode einer Transaktion:



Preclaiming ist unrealistisch und funktioniert nicht für alle Transaktionen.

Wait-for-graph (Warte-Graph)

Wait-for-graph (Warte-Graph)

Ein Wait-for-graph (Warte-Graph) $G = (V, E)$ besteht aus:

1. Der Menge der **laufenden** Transaktionen $V = \{T_1, \ldots, T_n\}$ als Knoten, sowie
2. Der Menge der gerichteten Kanten $E = \{(T_i, T_j)\}$, d.h. eine gerichtete Kante von T_i nach T_j, falls T_i auf einen Lock wartet, der durch T_j gehalten wird.

Erinnern Sie sich an den Konfliktgraphen aus Teil 1?
Genau. Das ist das gleiche in grün.

Zyklen im Warte-Graph

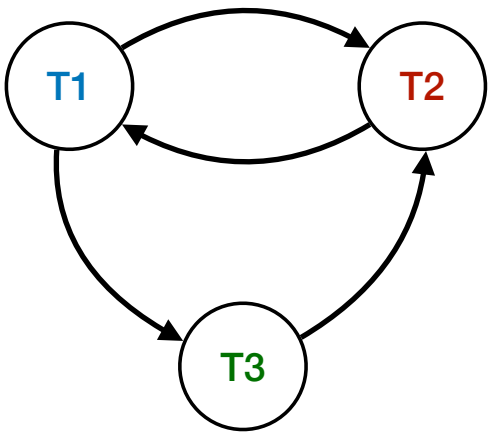
Ein Zyklus im Wartegraph zeigt einen Deadlock an.

Deadlock im Wait-for-graph

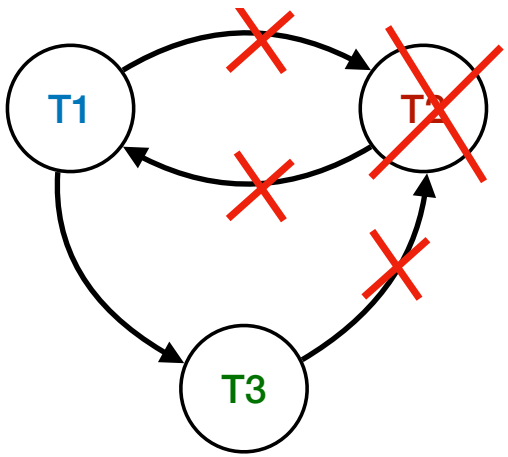
Deadlock im Wait-for-graph

Falls ein Zyklus im Wait-for-graph existiert, müssen solange Knoten (und alle mit ihnen verbundenen Kanten!) entfernt werden, bis kein Zyklus mehr vorhanden ist, d.h. der Graph zyklenfrei ist.
Knoten entfernen heißt dabei: die zugehörige Transaktionen abbrechen (Das Anwendungsprogramm kann sie dann erneut starten).

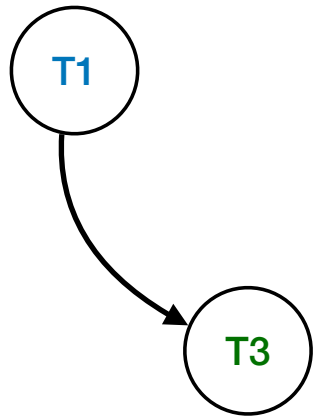
Beispiel:



Graph mit Zyklus



T2 abbrechen/entfernen



Ergebnis:
Graph ohne Zyklus

Konfliktserialisierbar $\neq$ `Serializable`

In der Datenbankwelt gibt es leider eine verwirrende Zahl ähnlich klingender aber unterschiedlicher Konzepte.

Konfliktserialisierbar $\not\Rightarrow$ `Serializable`

Der Ausführungsplan ist konfliktäquivalent zu einem seriellen Ausführungsplan.

Die Probleme damit:

1. Was, wenn nicht alle Transaktionen im Ausführungsplan committen? Dies verletzt potenziell die Isolation.
2. Was, wenn eine Anfrage nicht nur ein einzelnes Tupel betrifft?

Deswegen müssen wir mehr machen:

`Serializable` vs Serialisierbar

`Serializable`, perfekte Isolation

Die Isolationsstufe `Serializable` ist viel stärker als das theoretische Konzept „konfliktserialisierbar“ und verhindert auch Probleme, die durch abbrechende Transaktionen auftreten können. Dies gilt sowohl für einzelne Tupel als auch für Bereiche/Mengen von Tupeln, die zu Phantomproblemen führen können.

Serialisierbar...

...kann bedeuten:

1. Synonym für konfliktserialisierbar
2. Synonym für `Serializable`, perfekte Isolation

Isolationsstufen: Abschwächung der Konsistenzgarantien (isolation level)

Isolationsstufen: Abschwächung der Konsistenzgarantien (isolation level)

Viele Datenbanksysteme bieten die Möglichkeit an, die Isolation von Transaktionen gegeneinander abzuschwächen. Der Gewinn ist höhere Performanz, der Verlust: schwächere Isolationsgarantien, die (je nach Anwendung) zu Problemen mit der Datenbasis führen können.

- die Default-Einstellung von Datenbanksystemen ist typischerweise **nicht** perfekte Isolation (in Datenbanksprache: `SERIALIZABLE`)
- aber: perfekte Isolation sollte jedes Datenbanksystem implementieren (notfalls wird das intern übersetzt zu serieller Ausführung)
- was die schwächeren Isolationsstufen für Fehlersituationen zulassen, ist leider stark von der Implementierung der Nebenläufigkeitskomponente im Datenbanksystem abhängig.
- im Zweifel für das konkrete System sorgfältig die Doku lesen!

Beispiel für eine 2PL-basierte Realisierung von Isolationsstufen

look for "Isolation level" table on Google and description on Wikipedia

Isolationsstufe	**Read**	**Write**	**Probleme**
`Read Uncommitted`	keine Sperren	S2PL	Dirty Read, Cascading Rollback
`Read Committed`	kurzzeitige Sperren	S2PL	Non-repeatable Read
`Repeatable Read`	S2PL		Phantome
`Serializable` (perfekte Isolation)	S2PL mit Prädikatsperren		keine

Isolationsstufe in SQL setzen

```
BEGIN TRANSACTION ISOLATION LEVEL
{   SERIALIZABLE
  | REPEATABLE READ
  | READ COMMITTED
  | READ UNCOMMITTED
};
...
COMMIT;
```

Dies ist die PostgreSQL-Syntax. Die Syntax unterscheidet sich je nach Datenbanksystem.
In diesem Beispiel wird die Isolationsstufe für eine einzelne Transaktion angegeben.

Achtung:

Falls man nichts weiter angibt, ist in PostgreSQL die Default-Isolationsstufe: `READ COMMITTED`.

Grundsätzliche Extreme der Sperrverfahren

Schlecht:
Datenobjekte und Prädikate nicht sperren

$\Downarrow$

beliebig viele Probleme mit ACID und Deadlocks

Gut:
andere Transaktionen müssen niemals warten

$\Downarrow$

exzellente Performance

Isolationsstufe:
`READ UNCOMMITTED`

VS

Gut:
sehr restriktiv Datenobjekte und Prädikate sperren

$\Downarrow$

keine Probleme mit ACID und Deadlocks

Schlecht:
andere Transaktionen müssen oft unnötig warten

$\Downarrow$

schwache Performance

Isolationsstufe:
`SERIALIZABLE`

Isolationsstufen in Python simuliert

In [6]:
```
# Execute the given schedule using the transaction manager
tx_manager.execute_schedule(schedule, dump_exec_code=False)
```

```
******************
submitted_schedule
******************
0          TX2    =>  BEGIN()
1          TX2    =>  bal2_0 = READ(table_name=accounts, rowid=0, column=Balance)
2          TX2    =>  ASSERT(constraint=(bal2_0 >= 100))
3          TX1    =>  BEGIN()
4          TX1    =>  bal1_0 = READ(table_name=accounts, rowid=0, column=Balance)
5          TX2    =>  UPDATE(table_name=accounts, rowid=0, values={'Balance': bal2_0 - 100.0})
6          TX2    =>  COMMIT()
7          TX1    =>  ASSERT(constraint=(bal1_0 >= 100))
8          TX1    =>  bal1_0 = READ(table_name=accounts, rowid=0, column=Balance)
9          TX1    =>  UPDATE(table_name=accounts, rowid=0, values={'Balance': bal1_0 - 100.0})
10         TX3    =>  BEGIN()
11         TX3    =>  bal3_3 = READ(table_name=accounts, rowid=3, column=Balance)
12         TX3    =>  UPDATE(table_name=accounts, rowid=3, values={'Balance': bal3_3 + 100.0})
13         TX1    =>  bal1_3 = READ(table_name=accounts, rowid=3, column=Balance)
14         TX3    =>  ABORT()
15         TX1    =>  UPDATE(table_name=accounts, rowid=3, values={'Balance': bal1_3 + 100.0})
16         TX1    =>  COMMIT()

*****************
executed_schedule
*****************
0          TX2    =>  BEGIN()
1          TX2    =>  bal2_0 = READ(table_name=accounts, rowid=0, column=Balance)
2          TX2    =>  ASSERT(constraint=(bal2_0 >= 100))
5          TX2    =>  UPDATE(table_name=accounts, rowid=0, values={'Balance': bal2_0 - 100.0})
6          TX2    =>  COMMIT()
```

siehe github: Transaction Manager.ipynb

Handel, Banken, Ticketsystem

2. Was sind die Datenmanagement und -analyseprobleme dahinter?

...

Frage 1

Wie erlauben wir das nebenläufige Ändern von Daten, ohne dass fehlerhafte Daten entstehen?

Nebenläufigkeitskontrolle (Concurrency Control)

Frage 2

Wie entwerfen wir das so, dass das Verfahren und das resultierende Gesamtsystem effizient sind?

Beispielsweise durch S2PL mit Prädikatsperren bzw. abgeschwächte Garantien mittels Isolationsstufen.
Moderne DBMS benutzen MVCC oder eine Kombination mit S2PL (siehe Stammvorlesung).

Handel, Banken, Ticketsystem

4. 15 min: Transfer der Grundlagen auf die konkrete Anwendung

Geld abheben mit `READ COMMITTED`?

T1 : holt sich **kurze** Lesesperre: guckt, ob genug Geld auf dem Konto, nur wenn Kontostand größer als angefragter Betrag, darf Geld abgehoben werden; **gibt** Lesesperre wieder **zurück**!

T2 : holt sich **kurze** Lesesperre: guckt, ob genug Geld auf dem Konto, nur wenn Kontostand größer als angefragter Betrag, darf Geld abgehoben werden; **gibt** Lesesperre wieder **zurück**!

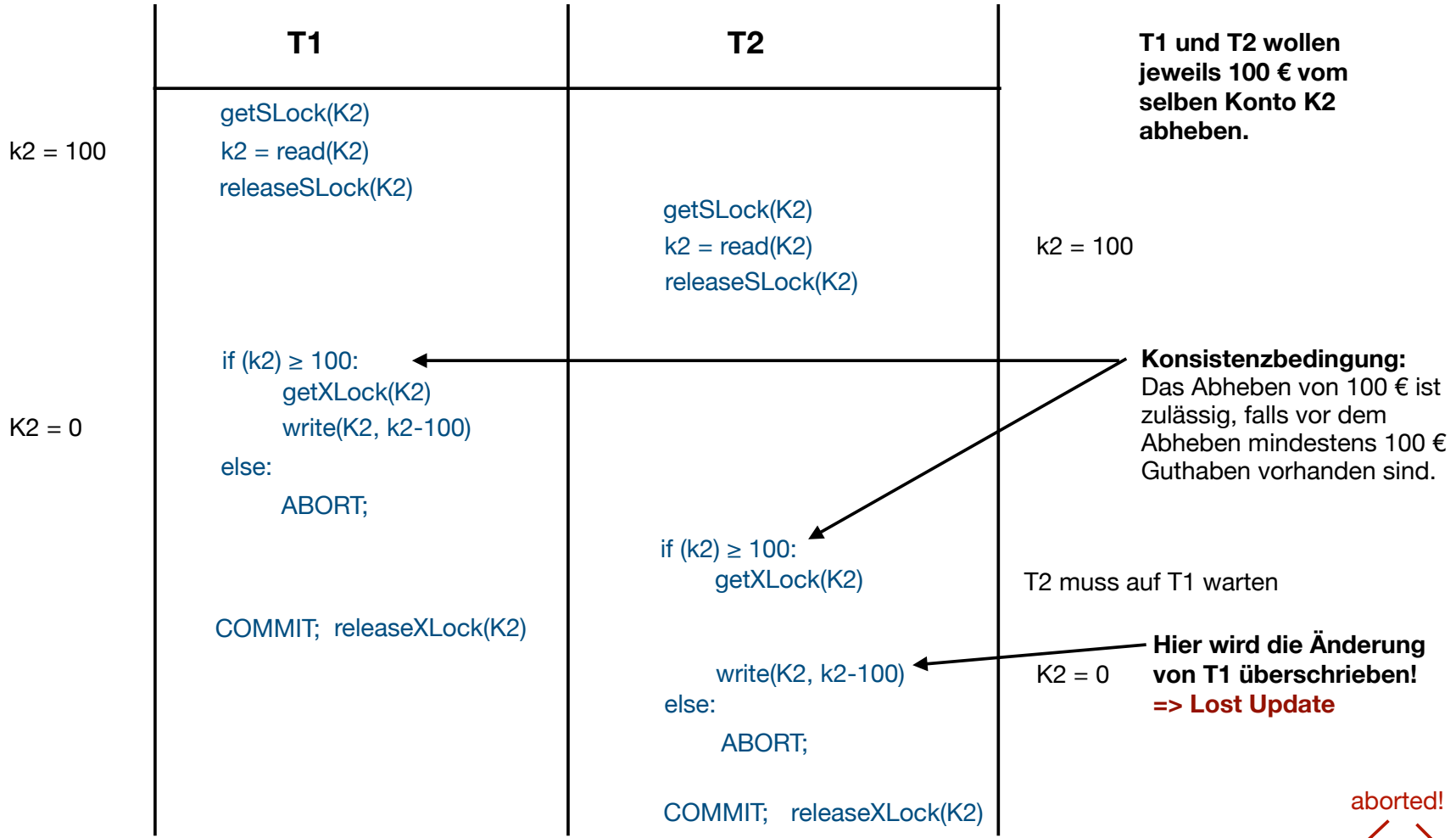
T2: bekommt **lange** Schreibsperre, verringert Kontostand und committed

T1: bekommt **lange** Schreibsperre, verringert Kontostand und committed

⇒ Verlorengegangene Änderung (Lost update)! Konsistenz aber nicht verletzt!

Für ein einfaches Abhebeszenario reicht `READ COMMITTED` **nicht** aus.

Schedule: Geld abheben mit READ COMMITTED?



T1 und T2 erfolgreich!
zweimal 100 € abgehoben!

=> Verletzung von I

vorher:

Konten	
Kontonummer	**Kontostand**
2	100 €

nachher:

Konten	
Kontonummer	**Kontostand**
2	0 €

SOLL:

aborted!

T1 →T2:

Konten
Kontostand
0 €

T2 →T1:

Konten
Kontostand
0 €

Geld abheben mit `REPEATABLE READ`?

T1 : holt sich **lange** Lesesperre: guckt, ob genug Geld auf dem Konto; nur wenn Kontostand größer als angefragter Betrag, darf Geld abgehoben werden

T2 : holt sich **lange** Lesesperre: guckt, ob genug Geld auf dem Konto; nur wenn Kontostand größer als angefragter Betrag, darf Geld abgehoben werden

T1: versucht Lesesperre zur Schreibsperre zu eskalieren, nicht möglich wegen T2, muss warten

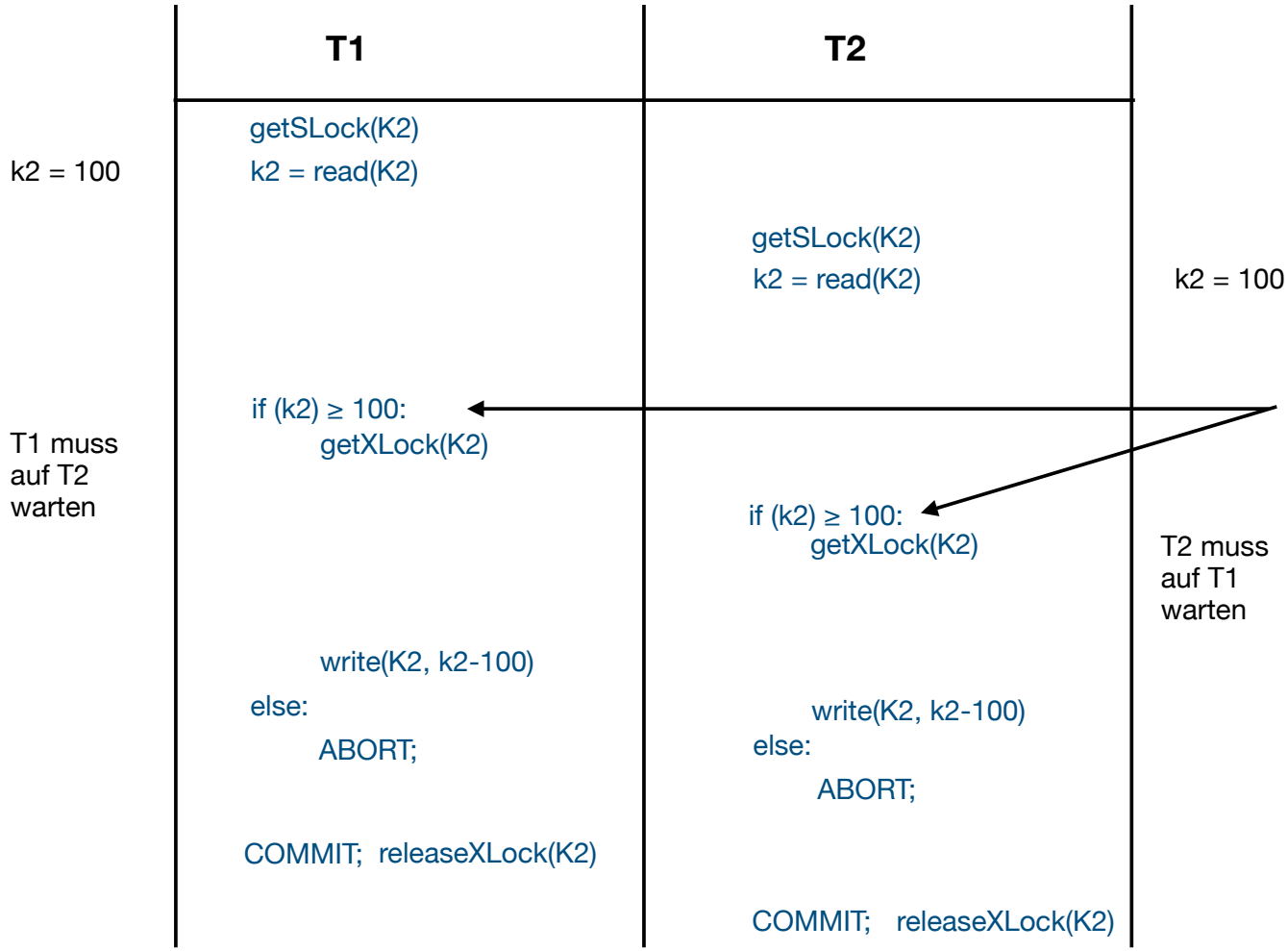
T2: versucht Lesesperre zur Schreibsperre zu eskalieren, nicht möglich wegen T1, muss warten

Deadlock...T1 oder T2 wird zurückgesetzt ... die andere Transaktion rechnet weiter

Für ein einfaches Abhebeszenario reicht `REPEATABLE READ` aus.

Wo reicht selbst diese Isolationsstufe `REPEATABLE READ` in einer Bank nicht aus?

Schedule: Geld abheben mit REPEATABLE READ?



T1 und T2 wollen jeweils 100 € vom selben Konto K2 abheben.

Konsistenzbedingung:
Das Abheben von 100 € ist zulässig, falls vor dem Abheben mindestens 100 € Guthaben vorhanden sind.

=> Deadlock

eine der beiden TAs wird aborted vom DBMS

die andere rechnet weiter

T1 oder T2 erfolgreich!
einmal 100 € abgehoben!

vorher:

Konten	
Kontonummer	**Kontostand**
2	100 €

nachher:

Konten	
Kontonummer	**Kontostand**
2	0 €

aborted!

SOLL:

T1 →T2:

Konten
Kontostand
0 €

T2 →T1:

Konten
Kontostand
0 €

Neues Konto anlegen mit REPEATABLE READ?

T1	T2
X = 1,5	X = SELECT avg(Zinssatz) FROM Konten WHERE PLZ = 66123;
	INSERT INTO KONTEN VALUES (9, 5000, 3, 66123); COMMIT;
X = wäre nach dem INSERT **eigentlich** 2 T1 benutzt aber weiterhin den alten Wert X = 1,5:	
UPDATE Konten SET Zinssatz=X WHERE PLZ = 66123; COMMIT;	

Hier wird von T2 ein neues Konto angelegt.
Dieses neue Tupel wird „Phantom“ genannt.

=> Verletzung von I

vorher:

KontenNeu			
Kontonummer	**Kontostand**	**Zinssatz**	**PLZ**
2	1000 €	1	66123
1	45 €	2	66117
7	2000 €	2	66123
8	74 €	2	66117
4	500 €	1	66119

nachher:

KontenNeu			
Kontonummer	**Kontostand**	**Zinssatz**	**PLZ**
2	1000 €	1,5	66123
1	45 €	2	66117
7	2000 €	1,5	66123
8	74 €	2	66117
4	500 €	1	66119
9	5000 €	1,5	66123

SOLL:

T1 →T2:

KontenNeu
Zinssatz
1,5
2
1,5
2
1
3

T2 →T1:

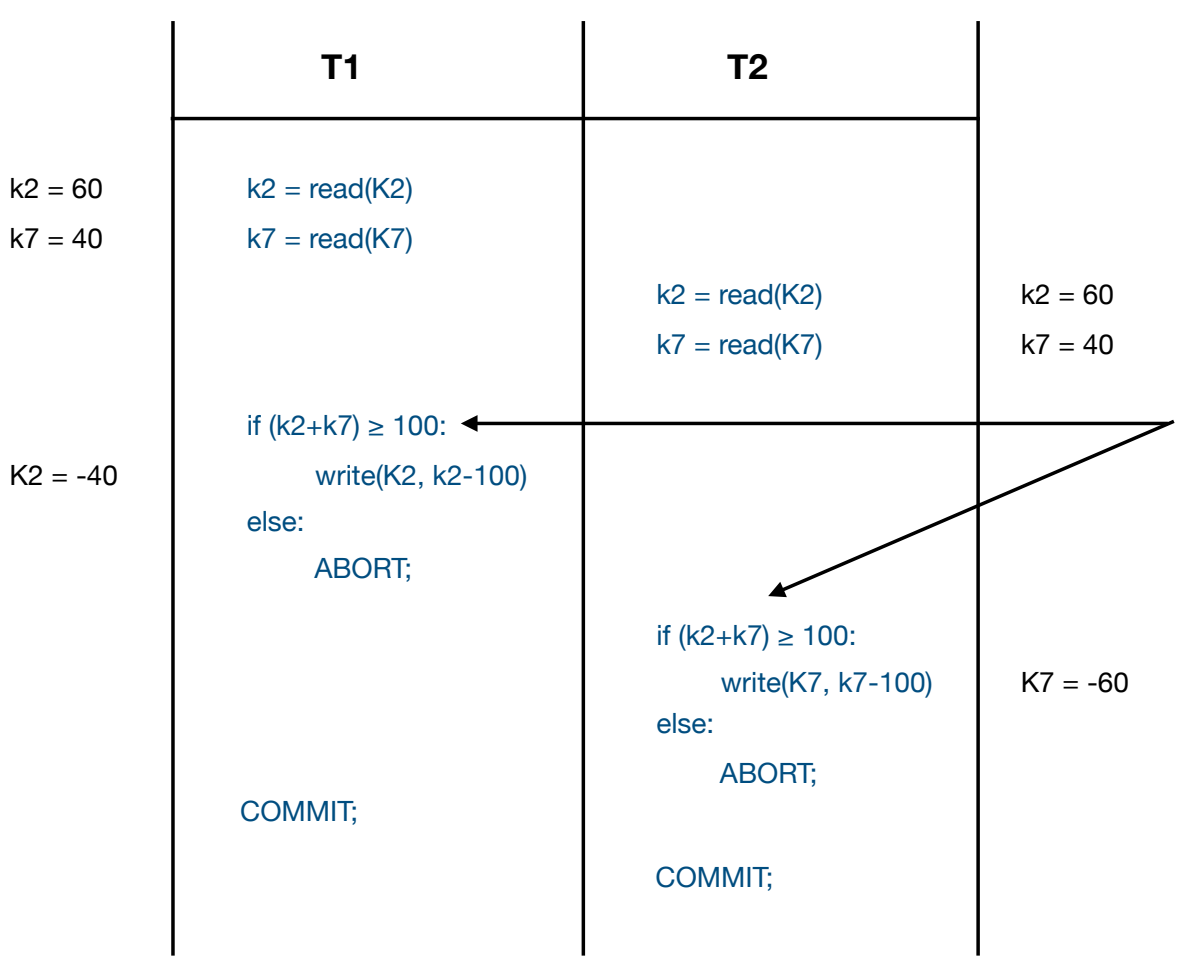
KontenNeu
Zinssatz
2
2
2
2
1
2

Neues Konto anlegen mit `SERIALIZABLE`?

Für ein Szenario, wo Änderungen basierend **auf einer Menge von Tupeln** getroffen werden, die durch WHERE selektiert werden, und die innerhalb einer Transaktion unterschiedliche Ergebnisse liefern können, brauchen wir `SERIALIZABLE`. Ansonsten reicht `REPEATABLE READ`.

Und jetzt machen wir aus der Uniwelt noch einen kurzen Ausflug in die Realität:

Uni vs Realität: Laufen diese Transaktionen durch?



T1 und T2 wollen jeweils 100 € von unterschiedlichen Konten abheben.

Konsistenzbedingung:
Das Abheben von 100 € ist zulässig, falls die Summe der beiden Konten vor dem Abheben mindestens 100 € beträgt! (also noch 100 € Guthaben vorhanden sind)

=> Verletzung von I und C

vorher:

Konten	
Kontonummer	**Kontostand**
2	60 €
7	40 €

Summe: 100 €

nachher:

Konten	
Kontonummer	**Kontostand**
2	-40 €
7	-60 €

Summe: -100 €

aborted!

SOLL:

T1 →T2:

Konten
Kontostand
-40 €
40 €

Summe: 0 €

T2 →T1:

Konten
Kontostand
60 €
-60 €

Summe: 0 €

Achtung: Uni

`READ COMMITTED`

leider ja..., warum?

Antwort: Wenn die Lesesperren nur kurz gehalten werden, können Schreibsperren erteilt werden, die beide Konten ändern dürfen.

`REPEATABLE READ`

nee, geht garnicht, warum?

Antwort: Die Lesesperren werden bis zum Ende der Transaktion gehalten. D.h. anderen Transaktionen dürfen Schreibsperren nicht erteilt werden, ein nebenläufiges Schreiben wird unmöglich für T2. Für diese Transaktion entsteht außerdem ein Deadlock.

`SERIALIZABLE`

nee, geht garnicht, warum?

Antwort: `SERIALIZABLE` enthält bereits alle Garantien von `REPEATABLE READ`...

Achtung: Realität

Läuft dieses Szenario in Oracle 12c Release 2 durch?

`SERIALIZABLE`

yep!

WTF?

- Dieses Problem heißt *write skew*.
- Es wird von den meisten DBMS erkannt, z.B. PostgreSQL und MS SQL Server.
- Dieses Problem tritt nur bei bestimmten Klassen von Algorithmen für die Nebenläufigkeitskontrolle auf: MVCC (multi-version concurrency control).
- Grundproblem: beide Transaktion arbeiten auf derselben Version der Datenbank (demselben *snapshot*) *und* ändern dann unterschiedliche Konten.
- Dasselbe Konto wäre kein Problem und würde entdeckt werden.
- Details? Siehe Stammvorlesung *Database Systems*

Zusammenfassung

Isolation

Die meisten Datenbanksysteme garantieren Atomarität und Isolation für Transaktionen. Durch nebenläufige Ausführung von Transaktionen wird die Leistung dieser Systeme enorm erhöht, ohne dadurch Probleme zu erzeugen.

Vorsicht mit Isolationsstufen

Isolationsstufen sind teilweise schwierig zu interpretieren. Im Zweifel immer die stärkere Isolationsstufe nutzen! Überprüfen Sie immer, welche Isolationsstufe per Default eingestellt ist und was das konkret im genutzten Datenbanksystem bedeutet!

Spickzettel für Teil 2

1. Zahlreiche Isolationsprobleme durch Lesen und/oder Schreiben „dreckiger“ oder nebenläufig veränderter Werte möglich
2. Lösung: tupelbasiertes Sperren, entweder kurzzeitig, langzeitig, 2PL, S2PL oder eine Mischung davon (siehe Isolationsstufen)
3. Phantom-Problem: Lesen über WHERE-Klausel kann innerhalb einer Transaktion potenziell unterschiedliche Ergebnisse liefern
4. Lösung: prädikatbasiertes Sperren von Bereichen der Relation
5. Deadlock-Problem: zwei oder mehr Transaktionen warten gegenseitig auf die Freigabe ihrer Sperren
6. Lösung: Wartegraph, um (Warte-)Zyklen erkennen zu können; Zyklen auflösen durch das gezielte Beenden von Transaktionen
7. Isolationsstufen zur Abschwächung der Isolationsgarantien: in vielen Systemen konfigurierbar;
8. der Default-Wert der Isolationsstufe in Datenbanksystemen ist meistens **nicht** `SERIALIZABLE`