

Handel, Banken, Ticketsystem (Teil 2)

VL Big Data Engineering
(aka Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

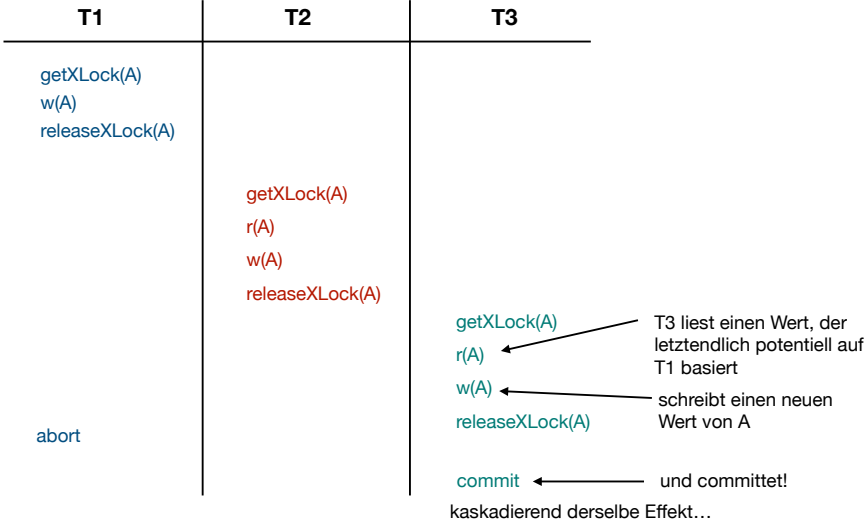
9. Oktober 2020

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. Sperren von Tupeln: Lese- und Schreibsperren, Kompatibilität von Sperren
5. Ausführungsplan (Historie): konkrete Verschränkung von Lese-/Schreiboperationen
6. Serieller Ausführungsplan: Transaktionen werden nacheinander ausgeführt
7. Konfliktserialisierbarer Ausführungsplan: äquivalent zu einem seriellen Ausführungsplan
8. Konflikt-Graph: aggregierte Form eines Ausführungsplan: Knoten entsprechen Transaktionen
9. Kein Zyklus im Konflikt-Graph \Rightarrow zugehöriger Ausführungsplan ist konfliktserialisierbar
10. 2PL hat zwei Phasen: Wachstumsphase (Sperren holen), Schrumpfphase (Sperren zurückgeben)
11. 2PL lässt nur konfliktserialisierbare Ausführungspläne zu
12. 2PL erlaubt aber leider kaskadierendes Rücksetzen

Kaskadierendes Zurücksetzen (cascading rollback, Wdh)

Beispiel: Dieser Ausführungsplan wird mit 2PL zugelassen (und ist konfliktserialisierbar)!



=> Verletzung von ACI

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

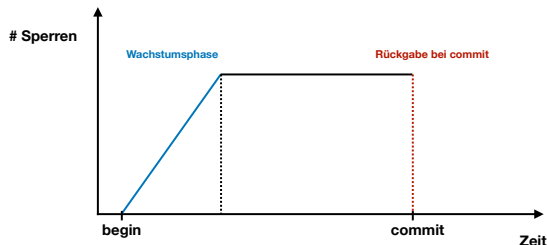
Striktes Two-Phase-Locking (Strict 2PL, S2PL)

Die Laufzeit einer Transaktion hat nur eine zeitliche Phase:

1. **Wachstumsphase:** Die Transaktion kann Sperren anfordern.

Sperren werden nicht vor Ende der Transaktion (durch commit oder abort) zurückgegeben.

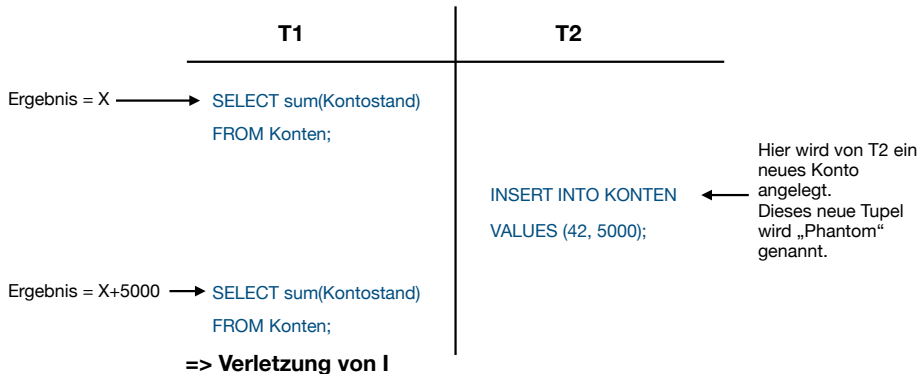
Zeitliche Übersicht für eine Transaktion unter S2PL:



S2PL verhindert Ausführungspläne mit kaskadierendem Zurücksetzen.

Phantomproblem

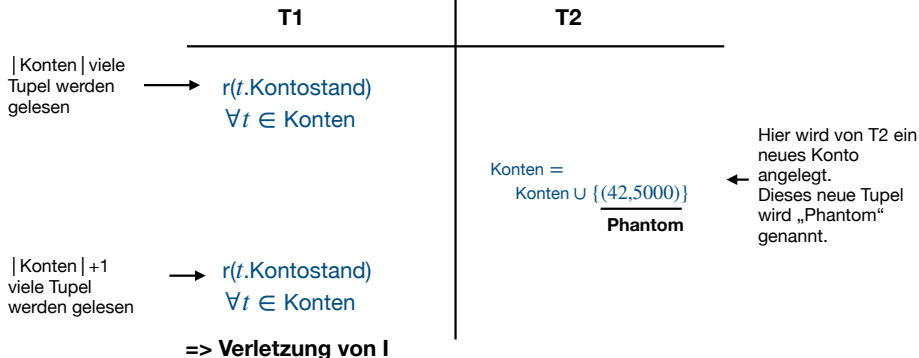
Beispiel: Dieser Ausführungsplan wird mit S2PL zugelassen!



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

Wir betrachten hier nicht mehr nur Lese-/Schreiboperationen, die **einzelne Tupel** lesen oder schreiben, wie bisher gefordert. Jetzt liest ein einzelnes SQL-Statement **mehr als ein Tupel!**

Sperrn 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 $\forall t \in \text{Konten } r(t.Kontostand)$, 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.

Prädikatsperren (Predicate Locking)

Prädikatsperren (Predicate Locking)

Um zu verhindern, dass eine Transaktion mit SQL-Statement, die auf mehrere Tupel zugreifen, ein Phantom sieht, müssen zusätzlich für solche Statements die Prädikate der WHERE-Bedingung dieser SQL-Statements gesperrt werden.

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!

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 DBMS ist typischerweise **nicht** SERIALIZABLE
- aber: SERIALIZABLE sollte jedes DBMS 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) Realisierung von Isolationsstufen

	Lesen	Schreiben	zusätzlich
Read Uncommitted	keine Sperren	S2PL	nichts
Read committed	kurzzeitige Sperren	S2PL	nichts
Repeatable Read	S2PL	S2PL	nichts
Serializable	S2PL	S2PL	Prädikatsperren

Zusammenfassung:

Konfliktserialisierbar \Rightarrow Serializable

Konfliktserialisierbarkeit impliziert Serialisierbarkeit im Sinne der obigen theoretischen Definition: Der Ausführungsplan ist äquivalent zu einem seriellen Ausführungsplan.

Das Problem damit: was, wenn nicht alle Transaktionen im Ausführungsplan committen? Deswegen müssen wir mehr machen:

Konfliktserialisierbar $\not\Rightarrow$ Serializable

Die Isolationsstufe Serializable ist viel stärker als das theoretische Konzept und verhindert auch Probleme, die durch abbrechende Transaktionen auftreten können.

Handel, Banken, Ticketsystem

4. Transfer der Grundlagen auf die konkrete Anwendung

...

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).

Zusammenfassung vom letzten Mal

- Konfliktserialisierbarkeit \Rightarrow "äquivalent zu einer seriellen Ausführung", \nRightarrow Atomarität!

„Konfliktserialisierbarkeit“ heißt: die verschränkte Ausführung der Aktionen ist zwar nicht das Problem. Das schließt aber nicht aus, dass es andere Probleme gibt!

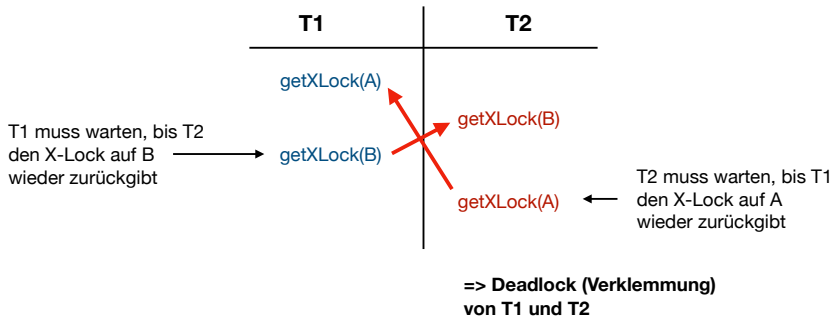
Konfliktserialisierbarkeit alleine ist nicht genug, um ACID zu garantieren!

- striktes 2 PL \Rightarrow Atomarität, aber Phantome und Deadlocks!!!

Deadlocks?

Verklemmung (Deadlock)

Beispiel: Dieser Ausführungsplan wird mit S2PL zugelassen!



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

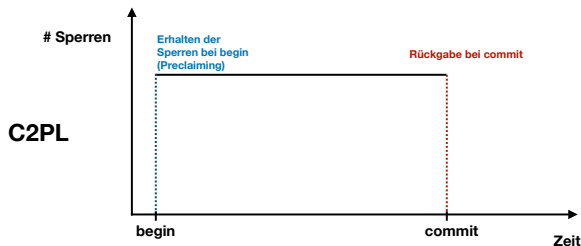
Konservatives Two-Phase-Locking (Conservative 2PL, C2PL, preclaiming)

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.

Beispiel

Beispielcode einer Transaktion:

Welche Tupel gelesen werden sollen, hängt hier von einer Bedingung im Applikationscode ab, hier: der Wert von X

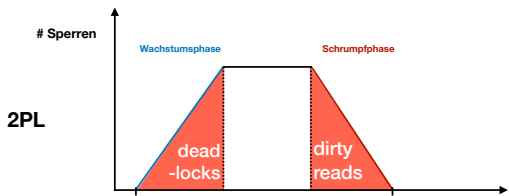
```
...
Falls X > 100:
    SELECT Kontostand
    FROM Konten
    WHERE Kontonummer = 43
Sonst:
    SELECT Kontostand
    FROM Konten
    WHERE Kontonummer = 42
...
```

Wir brauchen eine Lesesperre für das Tupel mit Kontonummer=43

Wir brauchen eine Lesesperre für das Tupel mit Kontonummer=42

Preclaiming ist unrealistisch und funktioniert nicht für alle Transaktionen.

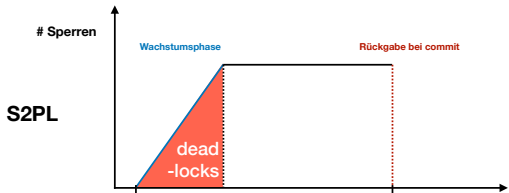
Übersicht über die verschiedenen 2PL-Varianten



deadlocks sind möglich

dirty reads durch andere Transaktionen sind möglich

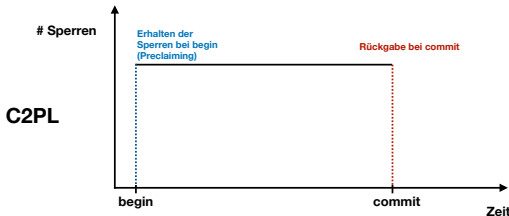
Sperren werden bei Bedarf angefordert und früh zurückgegeben



deadlocks sind möglich

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

Sperren werden bei Bedarf angefordert und bei commit zurückgegeben



deadlocks sind **nicht** möglich

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

Sperren werden bei begin angefordert und bei commit zurückgegeben

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

Lock Ordering

Einfaches Verfahren zur Verhinderung von Deadlocks:

Lock Ordering

1. Nummeriere (einmalig) alle Objekte, die gesperrt werden dürfen.
2. Jede Transaktion darf Sperren nur in der Reihenfolge dieser Nummerierung anfordern.
3. Benötigt eine Transaktion eine Sperre mit Nummer x , obwohl sie bereits eine Sperre mit Nummer $y > x$ hält, bricht sie ab.

Beispiel: (für eine Nummerierung von Objekten)

- Alle Relationen werden nach ihrem Namen angeordnet.
- Innerhalb jeder Relation kann jedes Tupel nach seinem Schlüssel angeordnet werden.
- D.h. wir haben nach zwei Kriterien sortiert.
- Eine solche Sortierung nennen wir *lexikographische Sortierung*.

Beispiel

Datenbasis:

Konten	
Kontonummer	Kontostand
2	1000 €
1	45 €
7	2000 €
8	74 €
4	500 €

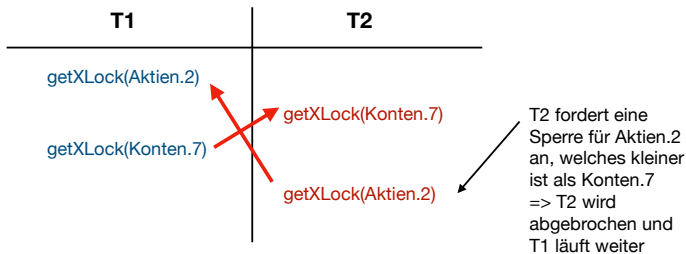
Aktien	
ID	Wert
5	88
2	23
4	23

Depots	
ID	Wert
1	45
5	23
6	67
4	11

Datenobjekte: Tupel

Nummerierung:

Aktien.2
Aktien.4
Aktien.5
Depots.1
Depots.4
Depots.5
Depots.6
Konten.1
Konten.2
Konten.4
Konten.7
Konten.8



Grundsätzliche Extreme der Sperrverfahren

Schlecht:

Datenobjekte und Prädikate
nicht sperren



beliebig viele Probleme mit
ACID und Deadlocks

Gut:

andere Transaktionen
müssen niemals warten



exzellente Performance

Isolationsstufe:

READ UNCOMMITTED

VS

Gut:

sehr restriktiv Datenobjekte
und Prädikate sperren



keine Probleme mit ACID
und Deadlocks

Schlecht:

andere Transaktionen
müssen oft unnötig warten



schwache Performance

Isolationsstufe:

SERIALIZABLE

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`.

Transaktionen in SQL (Transactions.ipynb)

Rollback Transactions

The next example shows a similar transaction as above. The only difference is that instead of making the changes persistent, we decide to `ABORT` the transaction by calling `rollback()` on the connection. It is equivalent to running the following transaction directly from the database shell:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id=3;
UPDATE accounts SET balance = balance + 100 WHERE id=1;
ABORT;
```

All changes performed by the aborted transaction will not be reflected by the database. Note that if we `close()` an open connection, `rollback()` will be performed implicitly.

```
In [6]: # Open a cursor to perform database operations
        cur1 = conn1.cursor()

        # Open a cursor on second connection to perform database operations
        cur2 = conn2.cursor()

        # Update balance of account 3, implicitly begins a transaction
        cur1.execute("""UPDATE accounts SET balance = balance - 100 WHERE id=3;""")

        # Update balance of account 1, implicitly begins a transaction
        cur1.execute("""UPDATE accounts SET balance = balance + 100 WHERE id=1;""")

        # Compare states visible to both transactions
        q_acc = """SELECT * FROM accounts WHERE id=1 OR id=3;"""
        cur1.execute(q_acc)
        cur2.execute(q_acc)
        print(f"Account balances observed by each transaction before COMMIT:\n\
              f"Transaction 1: {cur1.fetchall()}\n\
              f"Transaction 2: {cur2.fetchall()}\n")
```

[https://github.com/BigDataAnalyticsGroup/
bigdataengineering/blob/master/Transactions.ipynb](https://github.com/BigDataAnalyticsGroup/bigdataengineering/blob/master/Transactions.ipynb)

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

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**!

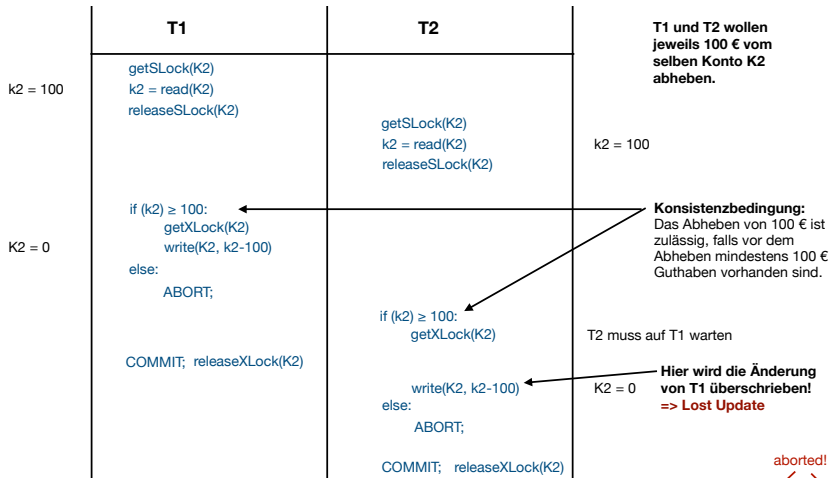
T2: bekommt **lange** Schreibsperre, verringert Kontostand und committed

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?

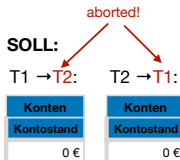


vorher: T1 und T2 erfolgreich!
zweimal 100 € abgehoben!

Konten	
Kontonummer	Kontostand
2	100 €

nachher: => Verletzung von I

Konten	
Kontonummer	Kontostand
2	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

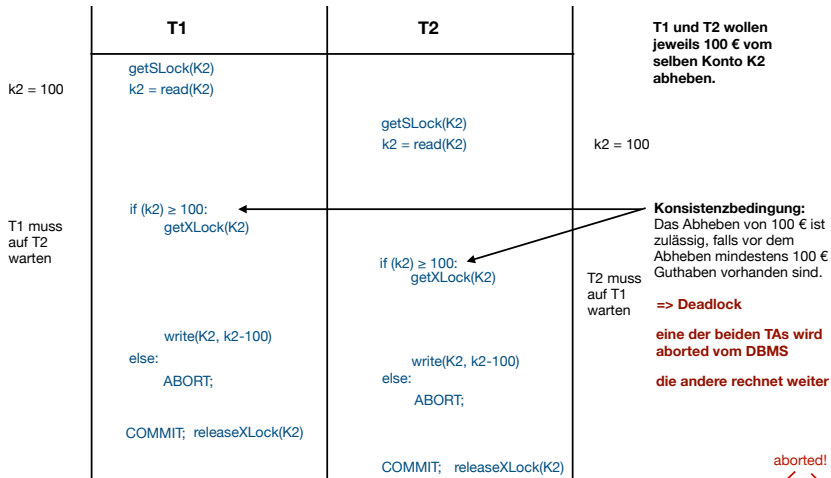
T2: versucht Lesesperre zur Schreibsperre zu eskalieren, nicht möglich wegen T1, muss warten

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?

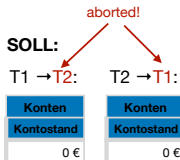


vorher: T1 oder T2 erfolgreich!
einmal 100 € abgehoben!

Konten	
Kontonummer	Kontostand
2	100 €

nachher:

Konten	
Kontonummer	Kontostand
2	0 €



Neues Konto anlegen mit REPEATABLE READ?

T1

```
X = 1,5  X = SELECT avg(Zinssatz)
        FROM Konten
        WHERE PLZ = 66123;
```

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;
```

T2

```
INSERT INTO KONTEN
VALUES (9, 5000, 3, 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:

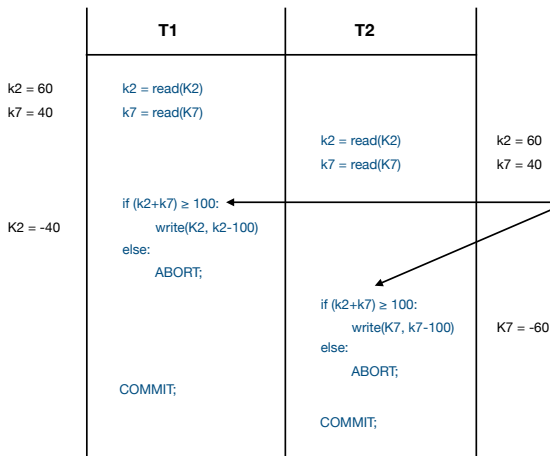
KontenNeu
Zinssatz
2
2
2
2
1
2

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 kann, 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:

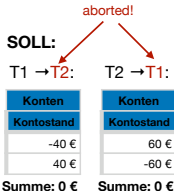
Konten	
Kontonummer	Kontostand
2	60 €
7	40 €

Summe: 100 €

nachher:

Konten	
Kontonummer	Kontostand
2	-40 €
7	-60 €

Summe: -100 €



Achtung: Uni

READ COMMITTED

leider ja..., warum?

Antwort: Wenn die Lesesperren nur kurz gehalten werden, können Schreibsperrern 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 Schreibsperrern nicht erteilt werden, ein nebenläufiges Schreiben wird unmöglich für T2.

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

Konfliktserialisierbar \Rightarrow Serialisierbarkeit

Konfliktserialisierbarkeit impliziert Serialisierbarkeit im Sinne der obigen theoretischen Definition: Der Ausführungsplan ist äquivalent zu einem seriellen Ausführungsplan. Mit anderen Worten: die Verschränkung der einzelnen Operationen im Ausführungsplan ist kein Problem.

Das Problem damit: was, wenn nicht alle Transaktionen im Ausführungsplan committen?

Deswegen müssen wir mehr machen:

Konfliktserialisierbar $\not\Rightarrow$ Serializable

Die Isolationsstufe `Serializable` ist viel stärker als das theoretische Konzept und verhindert auch Probleme, die durch abbrechende Transaktionen auftreten können.

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!