Handel, Banken, Ticketsystem (Teil 1)

VL Big Data Engineering
(aka Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

7. Juli 2020

Handel, Banken, Ticketsystem

Geplante Struktur für jeweils zwei Wochen Vorlesung:

1. Konkrete Anwendung: Handel, Banken, Ticketsystem
2. Was sind die Datenmanagement und -analyseprobleme dahinter?
3. Grundlagen, um diese Probleme lösen zu können
   (a) Folien
   (b) Jupyter/Python/SQL Hands-on
4. Transfer der Grundlagen auf die konkrete Anwendung

Handel, Banken, Ticketsystem

1. Konkrete Anwendung: Handel, Banken, Ticketsystem

ONLINE-BANKING

Zweite IT-Panne innerhalb kurzer Zeit – Commerzbank verärgert Kunden

Viele Commerzbank-Kunden haben am Freitag kein Online-Banking machen und kein Geld abheben können. Erst vor wenigen Wochen gab es einen ähnlichen Vorfall.

Andreas Kröner

24.08.2018 • Update: 24.08.2018 - 17:58 Uhr • Kommentieren • 2 x geteilt

22. Mai 2019, 05:11 Uhr Finanzindustrie

Software-Panne bei der Deutschen Bank

- Die Deutsche Bank hat eine IT-Panne bei einer Software entdeckt, mit der eigentlich Zahlungen von Großkunden überwacht werden sollten.
- Insidern zufolge waren bei diesem Programm jahrelang Parameter falsch programmiert.
- Die Deutsche Bank hat das Problem der Finanzaufsicht Bafin sowie der US-Notenbank Fed gemeldet. Man arbeite daran, "den Fehler schnellstmöglich zu beheben".

Technikpanne

Sparda-Bankautomaten ausgefallen - 3,6 Millionen Kunden betroffen

Wegen einer technischen Panne konnten Sparda-Kunden stundenlang kein Geld mehr den Filialautomaten ziehen. Auch Onlineüberweisungen funktionierten nicht. Sogar die Telefone des Instituts waren ausgefallen.

Logo der Sparda-Bank (Archiv)

Teilen Twittern E-Mail

herstag, 17.05.2018 19:00 Uhr Drucken Nutzungsrechte Feedback Kommentieren

Online-Bank N26

Pannen bei gefeierter Online-Bank

Von Barbara Schäder - 10. April 2019 - 19:18 Uhr

Die Smartphone-Bank N26 hat in nur drei Jahren zweieinhalb Millionen Kunden erobert. Doch nun hagelt es Kritik. Auch die Finanzaufsicht Bafin soll Verbesserungen angemahnt haben.

PROBLEME BEI ZAHLUNGSAUFTRÄGEN

Ärger über IT-Panne bei Commerzbank

VON HANNO MUSSLER - AKTUALISIERT AM 04.06.2019 - 14:53

Eine IT-Panne sorgt für Ärger bei Kunden der Commerzbank. Wegen einer technischen Störung konnten am Montag Daueraufträge, Überweisungen und Lastschriften nicht verarbeitet werden.

Datenverlust

Gravierende IT-Panne bei der UBS – bis zu 1500 Kunden betroffen

Mi 05.12.2018 - 10:01 Uhr | Aktualisiert 05.12.2018 - 10:01
von beat schmid, ch media

Ein IT-Fehler hat bei der UBS zu einer Datenpanne geführt. Dokumente von bis zu 1500 Kunden könnten verloren gegangen sein.

09.01.2019 | Unternehmen

Wieder IT-Probleme bei der Bank Austria

Nicht umsonst nimmt die FMA den Umgang der Banken mit der IT-Sicherheit 2019 besonders unter die Lupe. Die Dringlichkeit verdeutlicht ein schwerer Vorfall bei der Bank Austria.

Warum?

Die Gründe für diese Ausfälle sind vermutlich sehr unterschiedlich.

Aber ein paar Fragen kann man sich da als Informatiker schon mal stellen:

1. Wieso dauern die Ausfälle teilweise so lange?
2. Wieso dauert das Wiederherstellen eines konsistenten Zustands und das Wiederaufnehmen des Normalbetriebs teilweise so lange?
3. Wieso gibt es so häufig Ausfälle?
4. Wieso sind gleich immer so viele Kunden betroffen?
5. Wieso laufen diese Systeme angesichts der Investitionen nicht stabiler?

Wenn man mit Daten umgeht, gibt es zahlreiche mögliche (und typische) Fehlerquellen. Das ist seit Jahrzehnten bekannt.

<arrogant_mode>
Ach ja: die Lösungen auch....
</arrogant_mode>

Datenbankmanagementsysteme (DBMS)

Seit den 70er Jahren werden relationale Datenbanksysteme (RDBMS, meist nur DBMS) entwickelt. Moderne DBMS verfügen typischerweise über folgende Eigenschaften:

1. erweitertes relationales Model: JSON, Arrays, Text, räumliche Daten, etc.
2. sehr umfangreicher SQL-Dialekt (je nach System)
3. **automatischer** Anfrageoptimierer (regel- und kostenbasiert)
4. sehr hohe Performanz (meistens, je nach System)
5. massive Unterstützung für physisches Design (Indexstrukturen, Partitionierung, Caching, Materialisierung)
6. Unterstützung „moderner“ Hardware: DRAM, PRAM, FPGAs, GPUs, ...

Datenbankmanagementsysteme (DBMS)

Und was das Vermeiden und den Umgang mit Fehlersituationen umgeht insbesondere:

7. umfangreiche Unterstützung für Transaktionen und ACID
8. Concurrency Control (**automatische** Nebenläufigkeitskontrolle) **heute!**
9. **automatische** Crash-Recovery, Replikation, Backup
10. **automatische** Konsistenzkontrolle (leider oft nicht genutzt)
11. dynamische Sichten, Zugriffsrechte (logische Datenunabhängigkeit)

Stahlkocher, Thomas doerfer (CC BY-SA 3.0)

Zusammenfassung: Analogie ESP

vor 1997

Steuerung, die das Problem verhindert

+ unfassbar talentierte/r FahrerIn

nach 1997

+ automatisiertes Bremsen **einzelner** Räder

(ESP)

Pflicht seit 2014

Wichtige DBMS

Kommerziell:

- Oracle
- Microsoft
- DB2
- Actian Vector
- SAP Hana
- Exasol
- ...

Open Source:

- PostgreSQL
- MySQL
- SQLite
- ...

Gutes Verzeichnis (fast) aller Datenbanken:
Database of Databases, https://dbdb.io/

(bei Abruf am 30.6.2020: 708 Datenbanken)

Vorgehensweise in der Vorlesung

Bisher:

- erst: im Jupyter Notebook einmalig CSV-Dateien in einem Thread in Relationen laden
  (Python, SQLite, ...)
- dann: beliebig viele lesende Anfragen auf den Relationen!

  $\Rightarrow$ keine Probleme mit Nebenläufigkeit!

Jetzt:

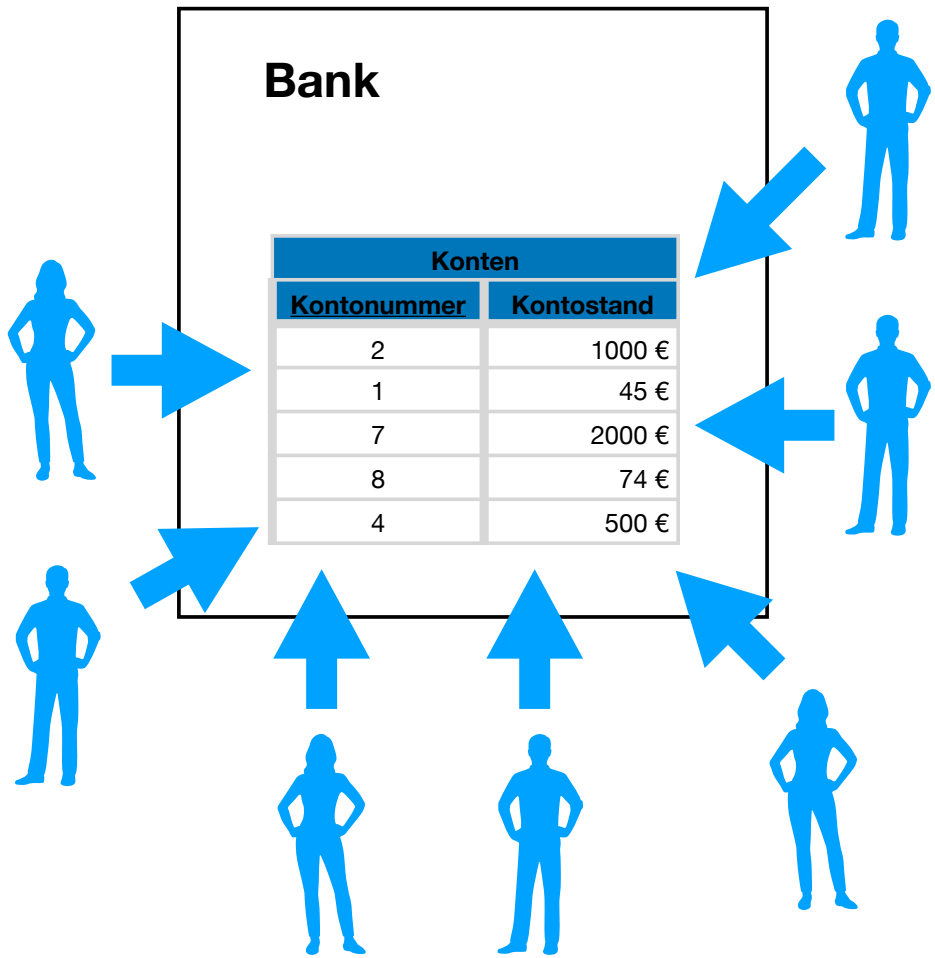
- Anfragen sollen Tupel in Relationen ändern dürfen, *wie sie wollen*.

  „*wie sie wollen*“? Hmmm...

Problemszenario

Problemszenario

Mehrere Kunden wollen gleichzeitig auf ihre Konten zugreifen, mindestens einer ändert den Kontostand durch Abheben, Einzahlen oder Überweisen.



Schauen wir uns mal verschiedene Varianten dieses Problemszenarios an:

Abheben und Einzahlen: Houston, we have (no) problem!

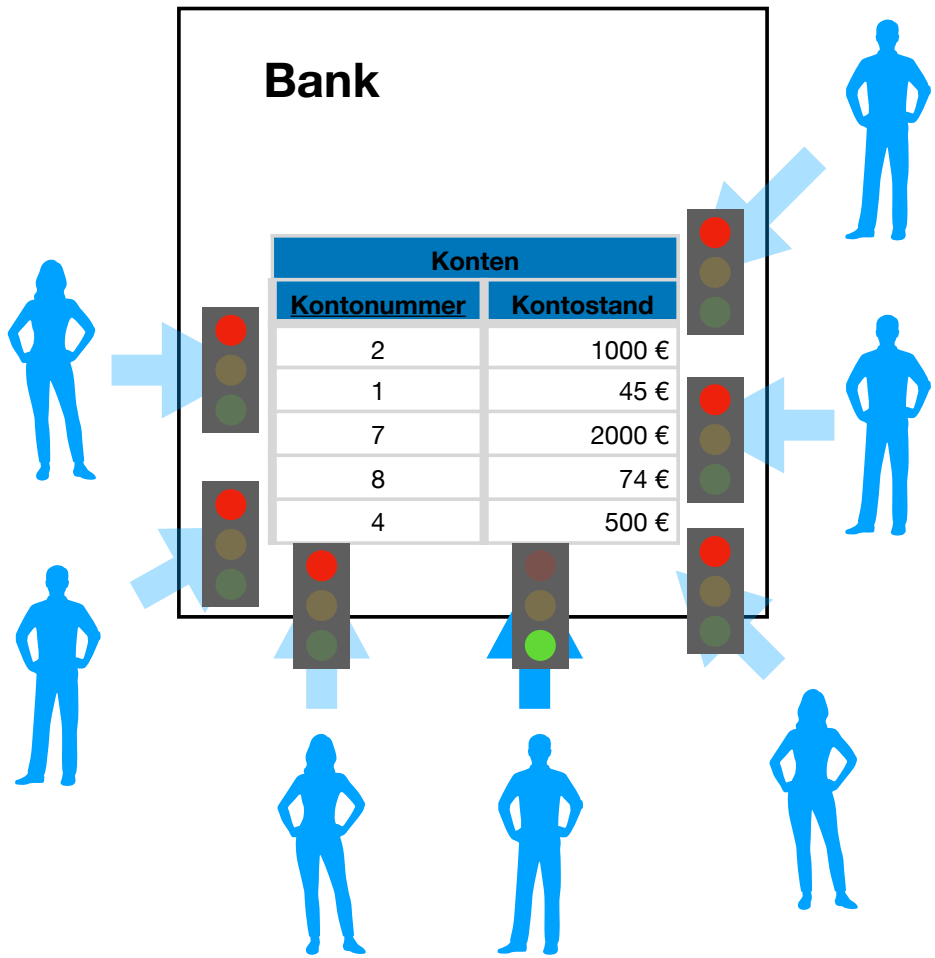
Kein Problemszenario, falls:

zu jedem Zeitpunkt maximal ein Kunde auf ein Tupel der Relation Konten zugreift

Lösung deshalb

Zugriff auf Relation für andere Kunden sperren

Das entspricht einer Warteschlange für alle Anfragen: serielle Abarbeitung aller Anfragen auf die Relation „Konten“.



Dem Kunden „gehört“ konzeptuell kurzzeitig die gesamte Bank!

Abheben und Einzahlen: Houston, we still have no problem!

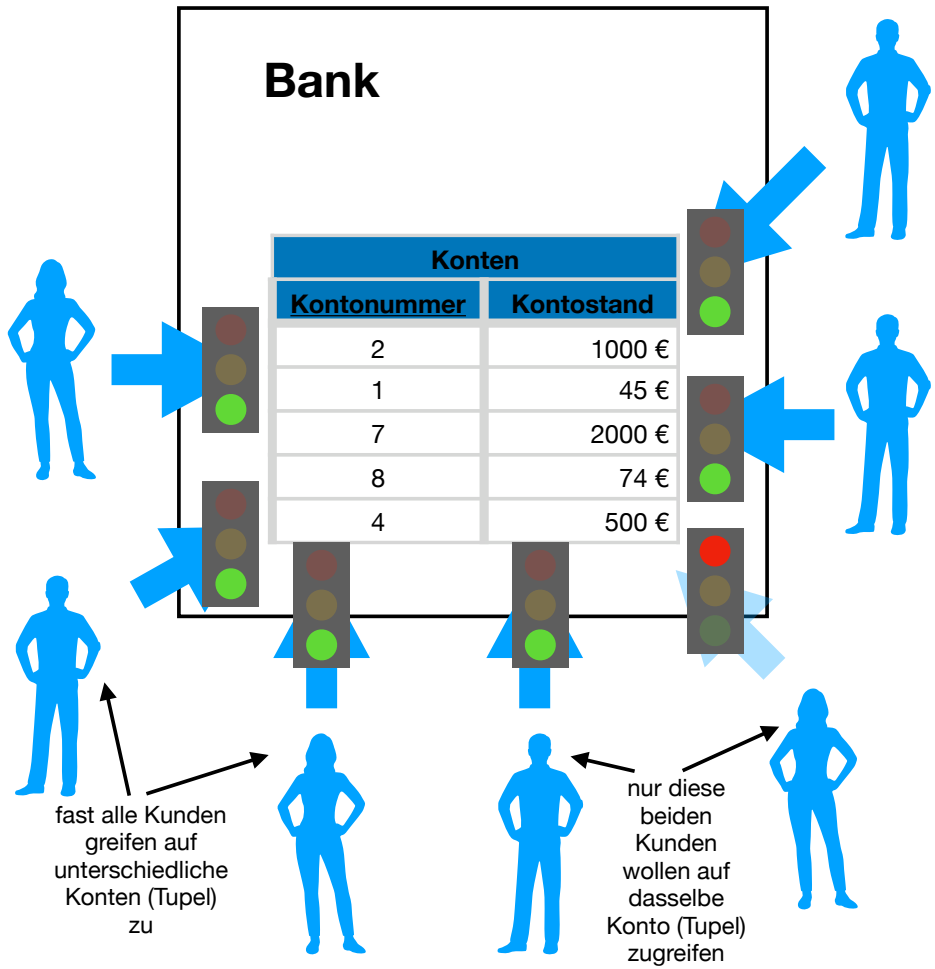
Kein Problemszenario, falls:

zu jedem Zeitpunkt maximal ein Kunde auf ein Tupel der Relation Konten zugreift

Lösung deshalb

Zugriff auf das jeweilige Tupel für andere Kunden sperren

Das entspricht einer Warteschlange für jedes Konto: serielle Abarbeitung aller Anfragen **pro Tupel/Konto**.



Dem Kunden „gehört“ konzeptuell kurzzeitig sein Konto!

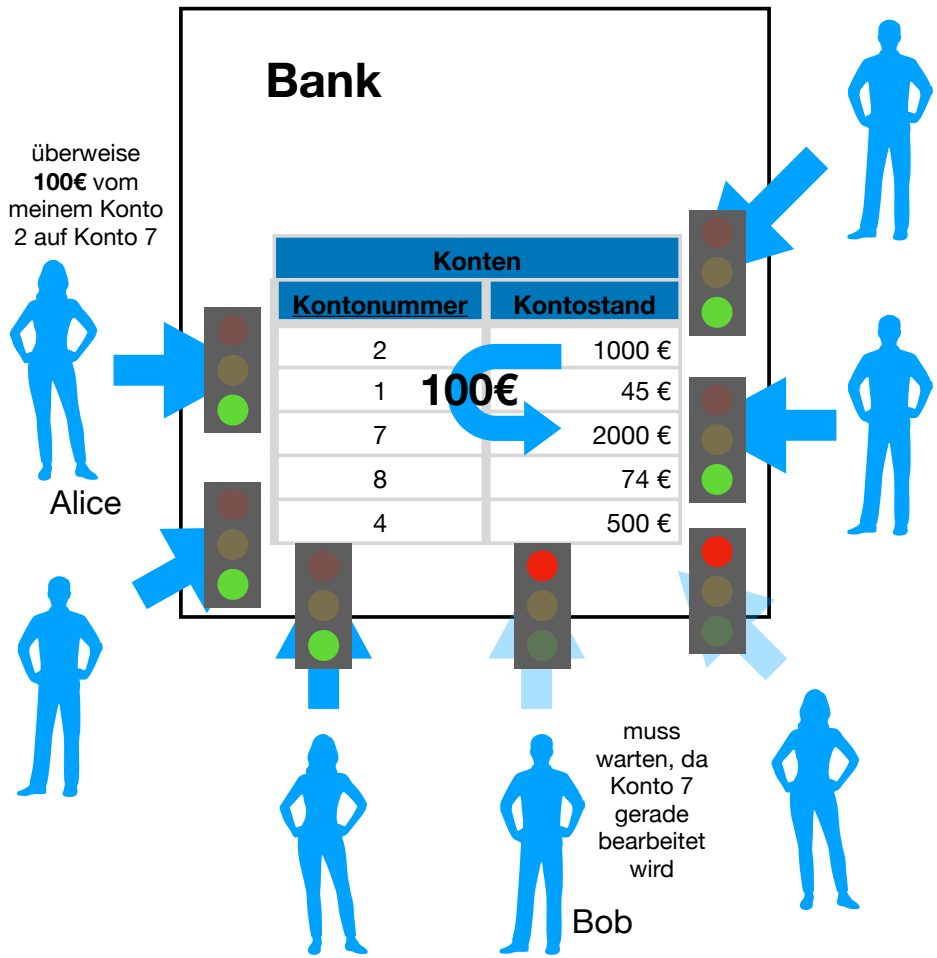
Überweisung: Houston, is this a problem?

Kein Problemszenario, falls:

die für die Überweisung notwendigen Tupel für alle anderen gesperrt werden

Lösung deshalb

Zugriff auf alle (meist nur zwei) Tupel der Überweisung für andere Kunden sperren



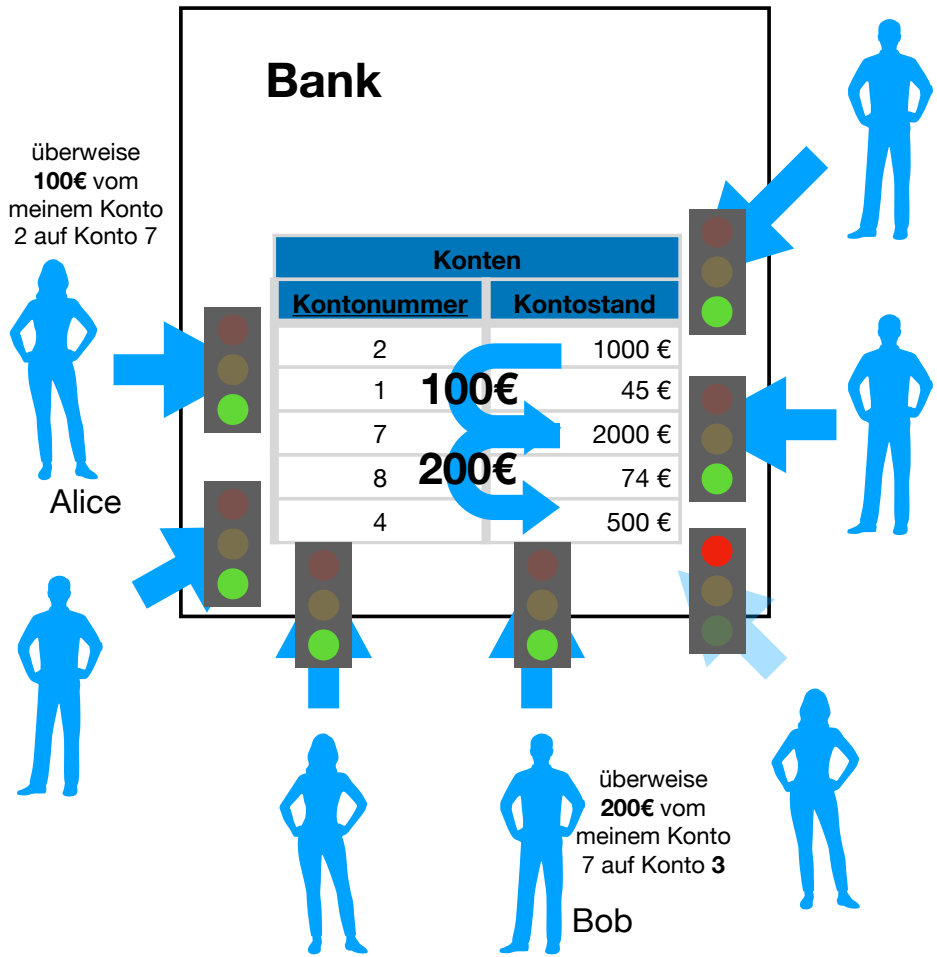
Zwei Überweisungen: Houston, we can handle this alone!

Kein Problemszenario, falls:

alle Tupel jeder Überweisung jeweils für alle anderen gesperrt werden

Lösung deshalb

Zugriff auf alle Tupel der Überweisung jeweils für andere Kunden sperren. Implizit führt dies zu einer Serialisierung (Festlegung der Reihenfolge) der Überweisungen.



Wie serialisieren wir diese Überweisungen, ohne für jede Überweisung jeweils die gesamte Relation zu sperren?

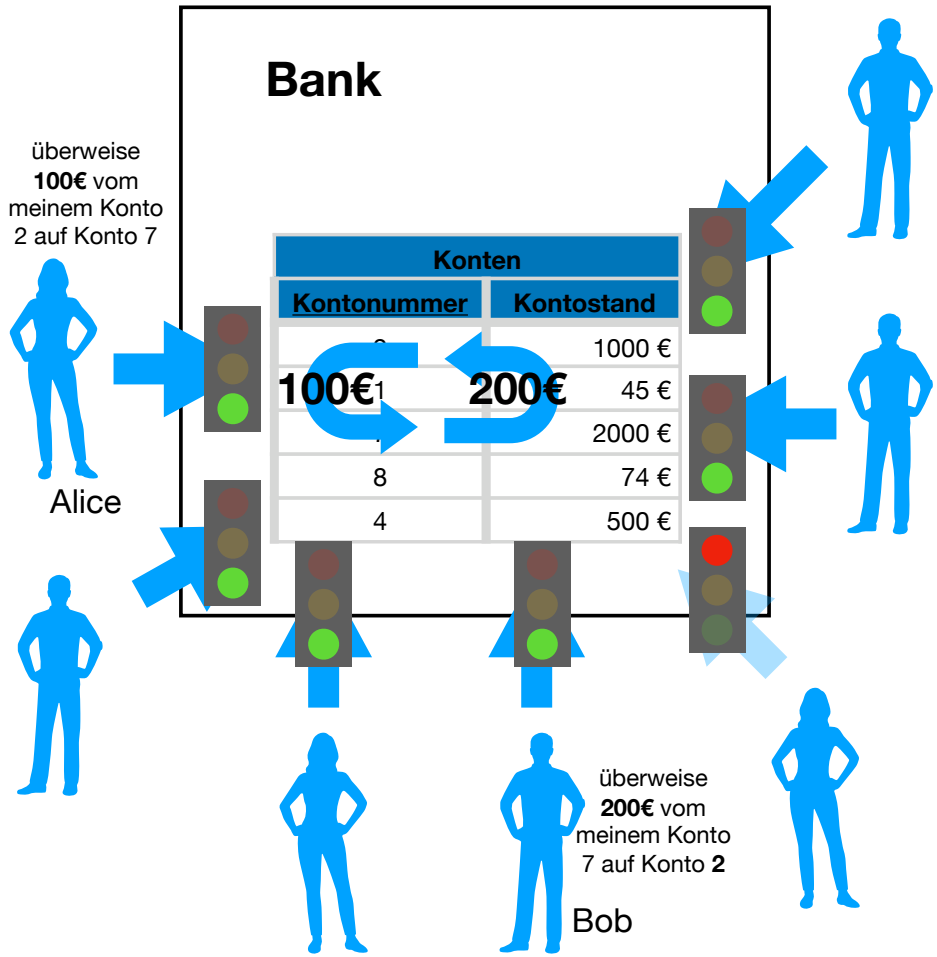
Zwei Überweisungen: Houston, we have a problem!

Kein Problemszenario, falls:

alle Tupel jeder Überweisung jeweils für alle anderen gesperrt werden.
UND: sich die beiden Überweisungen nicht verklemmen (Deadlock)

Lösung deshalb

???



Da gucken wir doch mal lieber im Detail rein...

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?

Frage 2

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

Handel, Banken, Ticketsystem

3. Grundlagen, um diese Probleme lösen zu können
   (a) Folien
   (b) Jupyter/Python/SQL Hands-on

Agenda:

Datenbanksysteme

Transaktion

ACID

Serialisierbarkeitstheorie:

- Ausführungsplan (Historie, Schedule)
- Serialisierbarkeitsgraph

Concurrency Control (Nebenläufigkeitskontrolle):

- Two-Phase-Locking (2PL)
- Isolationsstufe (isolation level)
- Multi-Version Concurrency Control (MVCC) (Stammvorlesung)

Leseoperation vs Schreiboperation

Leseoperation

Eine Leseoperation liest in einer Relation **maximal ein Tupel**.

Notation: $r(A)$: das Datenobjekt A (z.B. der Attributwert eines Tupels oder maximal das gesamte Tupel) wird gelesen.

Schreiboperation (auch Änderungsoperation)

Eine Schreiboperation verändert in einer Relation **maximal ein Tupel**. Es gibt drei unterschiedliche Arten von Schreiboperationen:

1. **Insert**: fügt ein neues Tupel in eine Relation ein.
2. **Update**: ändert ein, mehrere oder alle Attributwerte eines existierenden Tupels einer Relation.
3. **Delete**: entfernt ein existierendes Tupel aus einer Relation.

Notation: $w(A)$: das Datenobjekt A (z.B. der Attributwert eines Tupels oder auch maximal ein ganzes Tupel) wird geschrieben.

Transaktion

Transaktion

Eine Transaktion T bündelt eine oder mehrere Lese- und Schreiboperationen zu einer untrennbaren Einheit. Die Transaktion definiert die *konzeptuelle Reihenfolge*, in der ihre Lese- und Schreiboperationen ausgeführt werden sollen.

Notation: Der Beginn einer Transaktion wird mit b (für *begin*), das erfolgreiche Ende mit c (für *commit*, Deutsch: *festschreiben*) bzw. das erfolglose Ende mit a (für *abort*, Deutsch: *abbrechen*) markiert.
$x \to y$: „Operation x findet vor Operation y statt“. Jede Operation muss nach dem Beginn und vor dem Commit (oder Abort) erfolgen.

Beispiel:
Transaktion T_1 (Kontoüberweisung von Konto 2 nach Konto 7, s.o.):
K_2: Kontostand von Konto 2, K_7: Kontostand von Konto 7

$$b_1 \to \underbrace{r_1(K_2)}_{\text{alter Wert}} \to \underbrace{w_1(K_2)}_{\text{neuer Wert}} \to \underbrace{r_1(K_7)}_{\text{alter Wert}} \to \underbrace{w_1(K_7)}_{\text{neuer Wert}} \to c_1$$

Antwort: Die systemische Sichtweise

Achtung

Wir haben im letzten Beispiel **nirgendwo** notiert, dass der geschriebene Wert von K_2 und K_7 verändert wurde! Warum?

Statements der Host-Programmiersprache vs Lese-/Schreiboperationen in SQL

Software vermischt Statements der Host-Programmiersprache (Java, C++, Python, etc.) mit Lese- und Schreiboperationen (SQL). Das System, das diese SQL-Anfragen bearbeitet, sieht also nicht unbedingt die Statements der Programmiersprache.

- Deswegen nehmen wir im Folgenden verschärfend an, dass wir nur die Lese- und Schreiboperationen sehen.
- Mit anderen Worten: allein mit dieser Information müssen wir in der Lage sein, zu entscheiden, ob bestimmte Lese- und Schreiboperationen erlaubt sind oder nicht.

Atomarität (Atomicity, A)

Atomarität (Atomicity, A)

Eine Transaktion ist eine untrennbare Einheit. Die Lese- und Schreiboperationen einer Transaktion werden entweder vollständig oder gar nicht ausgeführt.

Beispiele:
vollständig ausgeführte Transaktion von Alice (überweise 100€ von Konto 2 nach Konto 7, **committer**):

$b_1 \rightarrow r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow w_1(K_7) \rightarrow c_1$

teilweise ausgeführte Transaktion (**aborted**):

$b_1 \rightarrow r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow a_1$

Hier wurde nur $w_1(K_2)$ ausgeführt aber nicht mehr $w_1(K_7)$. Wir müssen sicherstellen, dass diese partiellen Änderungen nicht in die Datenbasis eingebracht werden.

Atomarität impliziert:

Keine halben ~~Sachen~~ Transaktionen

Die Datenbasis (Menge der Relationen) muss davor geschützt werden, dass nur Teile von Änderungen einer Transaktion in die Datenbasis eingebracht werden oder auch nur sichtbar werden für andere Transaktionen, bevor die Transaktion committer!

Wenn wir eine Schreiboperation an die Datenbasis schicken, darf diese **nicht** direkt sichtbar werden für andere Transaktionen!

Konsistenz (Consistency, C)

Konsistenz (Consistency, C)

Eine Transaktion T hinterlässt die Datenbasis beim Committen in einem konsistenten Zustand. Während der Abarbeitung der Transaktion T darf die Datenbasis beliebig inkonsistent sein (**aus Sicht von T!**).

Konsistenzbedingung

Eine Konsistenzbedingung ist eine Invariante an die Datenbasis. Sie legt eine Bedingung an die Datenbasis fest.

Beispiel:
Konsistenzbedingung: für Überweisungen innerhalb einer Bank gilt, dass *die Summe über alle Kontostände immer gleich ist.*

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

Dies ergibt **immer** dasselbe Ergebnis, **vor und nach** Ausführung einer Überweisung innerhalb der Bank.

Konsistenzbedingungen

Beispiele:

- Schlüssel:
  `PRIMARY KEY`
- Fremdschlüssel existiert:
  `FOREIGN KEY (Person_ID) REFERENCES Persons(ID)`
- ...

Konsistenz impliziert:

Konsistenzbedingungen für jede Transaktion überprüfen!

Die Datenbasis (Menge der Relationen) muss davor geschützt werden, dass Änderungen einer Transaktion in die Datenbasis eingebracht werden, die irgendeine Konsistenzbedingung verletzen. D.h. bevor eine Transaktion committen darf, müssen wir sicherstellen, dass die von dieser Transaktion vorgeschlagenen Änderungen keine dieser Konsistenzbedingungen verletzt.

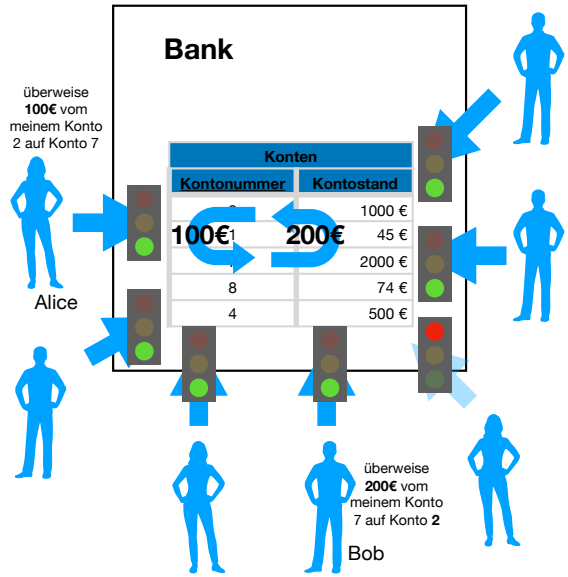
Isolation (Isolation, I)

Isolation (Isolation, I)

Nebenläufig ausgeführte Transaktionen beeinflussen sich nicht. Für jede Transaktion T_i sieht es so aus, als wäre T_i die einzige Transaktion, die ausgeführt wird. Was andere Transaktionen $T_{j \neq i}$ machen bzw. deren Effekte auf die Datenbasis, ist für T_i nicht sichtbar.

Beispiel:

Wie auch immer diese Überweisungen ausgeführt werden:
nacheinander (in welcher Reihenfolge auch immer) oder
nebenläufig (die einzelnen Lese- und Schreiboperationen der Überweisungen werden irgendwie verzahnt ausgeführt):



Die Weltsicht jeder einzelnen Transaktion ist: ich bin die einzige Transaktion, die aktuell ausgeführt wird.

Isolation impliziert:

Nebenläufigkeit nicht dem Zufall überlassen!

Die Datenbasis (Menge der Relationen) muss davor geschützt werden, dass nebenläufig ausgeführte Transaktionen die Eigenschaften I oder C verletzen. D.h. wir dürfen nicht jeder Transaktion freien Zugriff auf die Datenbasis gewähren, sondern müssen die Datenbasis vor solchen Änderungen schützen.

Das schauen wir gleich etwas mehr im Detail an.

Dauerhaftigkeit (Durability, D)

Dauerhaftigkeit (Durability, D)

Falls eine Transaktion T committer, müssen die Effekte ihrer Schreiboperationen (also die von ihr eingebrachten Effekte von Änderungsoperationen) in der Datenbasis erhalten bleiben, egal was passiert.

Beispiel:

1. Überweisungstransaktion committet,
2. Dann gibt es einen Stromausfall.

Wurden die Änderungen vor dem Stromausfall auf Festplatte/SSD persistiert? Und wenn ja, welche? Und wie stellen wir dies fest?

Wurden möglicherweise nur Teile der Änderungen persistiert?
⇒ zusätzliches Problem mit Atomarität (A) und Konsistenz (C)?

Hmmmm....

Dauerhaftigkeit impliziert:

Wir müssen die Datenbasis gegen Fehler jeder Art schützen

Algorithmen und Systeme sind so zu konzipieren, dass bei Fehlern jeder Art (vom Software-Fehler bis zu katastrophalen Fehlern), D garantiert ist.

Das ist nicht ganz einfach. Lässt sich aber machen.

Und was ist überhaupt ein „katastrophaler Fehler“?

Dieses Thema schauen wir uns im Detail in der Stammvorlesung „Database Systems“ an.

Serialisierbarkeitstheorie: Ausführungsplan (Schedule)

Ausführungsplan (auch Historie; Schedule)

Ein Ausführungsplan AP legt für eine Menge von Transaktionen $T_1, \ldots, T_n$ die Reihenfolge ihrer Lese-, Schreib- und Commit- bzw. Abbruch-Operationen fest.
Dabei gilt, dass die Reihenfolge von Lese- und Schreiboperationen innerhalb einer Transaktion immer erhalten bleibt. Für eine Historie muss keine totale Ordnung definiert werden **mindestens aber** die Reihenfolge der **Konfliktoperationen**.

Notation:
Den Beginn jeder Transaktion markieren wir im Ausführungsplan nicht explizit (oben hatten wir dies für einzelne Transaktionen mit b_i notiert): in einem Ausführungsplan markiert die erste Operation einer Transaktion jeweils implizit den Beginn dieser Transaktion.

Konfliktoperationen

Konfliktoperationen

Zwei Lese-/Schreiboperationen eines Ausführungsplans heißen Konfliktoperationen, falls **alle** folgenden Bedingungen gelten:

1. sie gehören zu unterschiedlichen Transaktionen,
2. beide greifen auf dasselbe Datenobjekt zu,
3. mindestens eine von ihnen ist eine Schreiboperation.

Beispiele:

$r_1(A)$ und $r_2(A)$: keine Konfliktoperationen, nur Leseoperationen
$r_1(A)$ und $w_1(A)$: keine Konfliktoperationen, dieselbe Transaktion
$r_1(A)$ und $w_2(A)$: Konfliktoperationen, alle Bedingungen erfüllt
$w_1(A)$ und $w_2(A)$: Konfliktoperationen, alle Bedingungen erfüllt
$w_1(A)$ und $w_2(B)$: keine Konfliktoperationen, nicht dasselbe Datenobjekt.

Ausführungsplan mit totaler Ordnung

Ausführungsplan mit totaler Ordnung

Ein Ausführungsplan mit totaler Ordnung legt für jedes Paar von Lese-, Schreib- und Commit- bzw. Abbruch-Operationen in S die Reihenfolge fest.

Beispiel:

Transaktion von Alice (überweise 100€ von Konto 2 nach Konto 7):

$r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow w_1(K_7) \rightarrow c_1$

Transaktion von Bob (überweise 100€ von Konto 7 nach Konto 2):

$r_2(K_7) \rightarrow w_2(K_7) \rightarrow r_2(K_2) \rightarrow w_2(K_2) \rightarrow c_2$

Ein möglicher Ausführungsplan mit **totaler Ordnung** hierfür:

$r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow r_2(K_7) \rightarrow w_1(K_7) \rightarrow c_1 \rightarrow w_2(K_7) \rightarrow r_2(K_2) \rightarrow w_2(K_2) \rightarrow c_2$

Ausführungsplan mit partieller Ordnung

Ausführungsplan mit partieller Ordnung

Ein Ausführungsplan mit partieller Ordnung zwingt die Operationen nicht in eine totale Ordnung, legt aber **mindestens** für jedes Paar von **Konfliktoperationen** in S die Reihenfolge fest.

Beispiel:

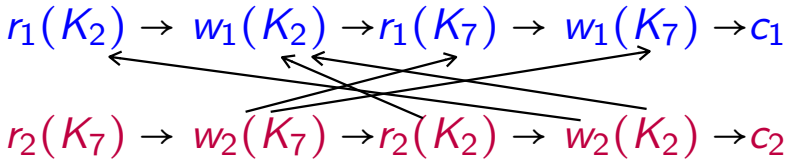
$$\begin{array}{llll} r_1(A) \rightarrow & w_1(A) \rightarrow & w_1(B) \rightarrow & c_1 \\ & & \uparrow & \\ & r_2(B) \rightarrow & w_2(B) \rightarrow & r_2(C) \rightarrow \quad c_2 \end{array}$$

In diesem Ausführungsplan wurde der Pfeil $r_2(B) \rightarrow w_1(B)$ weggelassen (der nach unserer Definition eigentlich spezifiziert werden müsste).

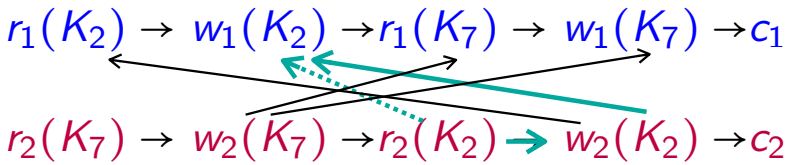
Es gilt aber $r_2(B) \rightarrow w_2(B)$ und $w_2(B) \rightarrow w_1(B)$

$\Rightarrow r_2(B) \rightarrow w_1(B)$.

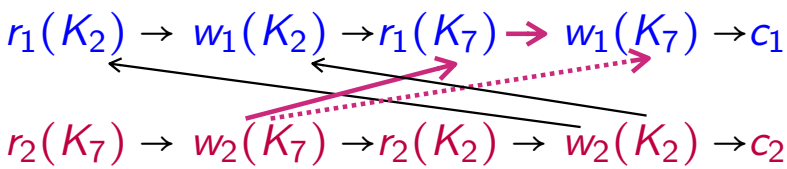
Beispiel für die Vereinfachung eines Ausführungsplans



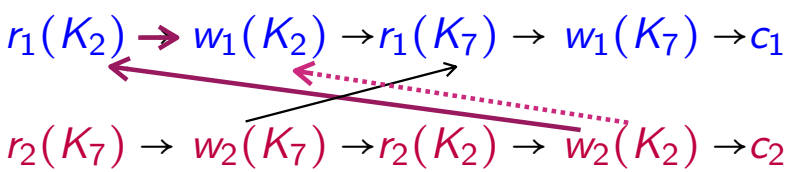
überall Spaghetti



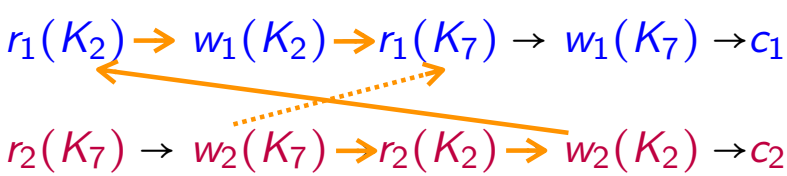
die gestrichelte Kante ist redundant, denn innerhalb von T_2 muss $r_2(K_2)$ sowieso vor $w_2(K_2)$ ausgeführt werden!



die gestrichelte Kante ist redundant, denn innerhalb von T_1 muss $r_1(K_7)$ sowieso vor $w_1(K_7)$ ausgeführt werden!



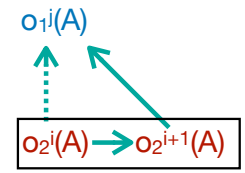
die gestrichelte Kante ist redundant, denn innerhalb von T_1 muss $r_1(K_2)$ sowieso vor $w_1(K_2)$ ausgeführt werden



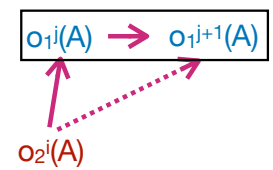
die gestrichelte Kante ist redundant, denn innerhalb von T_2 muss $w_2(K_7)$ sowieso vor $w_2(K_2)$ ausgeführt werden und innerhalb von T_1 muss $r_1(K_2)$ vor $r_1(K_7)$ ausgeführt werden

gestrichelte Kanten sind redundant und werden jeweils entfernt:

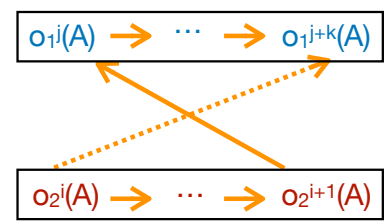
Fall 1



Fall 2



Fall 3



Seriell (serial)

Seriell (serial)

Ein Ausführungsplan AP heißt **seriell**, falls alle Transaktionen in AP vollständig nacheinander ausgeführt werden. Mit anderen Worten: zu jedem Zeitpunkt ist nur eine Transaktion aktiv und führt alle ihre Operationen vollständig aus.

Anzahl der seriellen Ausführungspläne

Für n Transaktionen gibt es $n!$ mögliche serielle Ausführungspläne.

Beispiele:

$r_1(A) \rightarrow w_1(A) \rightarrow c_1 \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2$
ist seriell

$r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow r_1(A) \rightarrow w_1(A) \rightarrow c_1$
ist seriell

$r_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$
ist **nicht** seriell

Konfliktäquivalent

Konfliktäquivalent

Zwei Ausführungspläne AP1 und AP2 heißen **konfliktäquivalent**, falls alle Konfliktoperationen in AP1 und AP2 in derselben Reihenfolge spezifiziert sind.

Notation: AP1 $\equiv$ AP2

Beispiele:

AP1: $r_1(A) \rightarrow r_2(B) \rightarrow w_1(A) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$
AP2: $r_1(A) \rightarrow w_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$
$\Rightarrow$ AP1 $\equiv$ AP2

AP3: $r_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$
Die Konfliktoperationen $w_1(A)$ und $r_2(A)$ wurden getauscht!
$\Rightarrow$ AP1 $\not\equiv$ AP3 und AP2 $\not\equiv$ AP3.

Achtung

Die relative Reihenfolge der Operationen innerhalb einer Transaktion wird (weiterhin) nicht verändert!

Konfliktserialisierbar

Konfliktserialisierbar

Ein Ausführungsplan AP heißt **konfliktserialisierbar**, falls er konfliktäquivalent zu einem seriellen Ausführungsplan ist.

Vertauschen von Nicht-Konfliktoperationen

Lässt sich ein beliebiger Ausführungsplan AP durch **Vertauschen von Nicht-Konfliktoperationen** in einen seriellen Ausführungsplan überführen, so ist AP konfliktserialisierbar.

Beispiele:

AP: $r_1(A) \rightarrow r_2(B) \rightarrow w_1(A) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$

kann durch Vertauschen überführt werden in:
$r_1(A) \rightarrow w_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$
und dann:
$r_1(A) \rightarrow w_1(A) \rightarrow c_1 \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2$

Da dies ein serieller Ausführungsplan ist $\Rightarrow$ AP ist konfliktserialisierbar.

Konflikt-Graph (Precedence Graph)

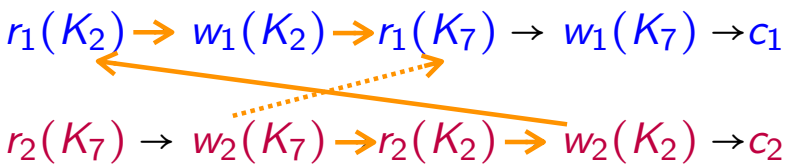
Konflikt-Graph (Precedence Graph)

Gegeben ein Ausführungsplan AP. Ein Konflikt-Graph (Precedence Graph) für AP hat einen Knoten für jede Transaktion in AP sowie eine gerichtete Kante für jedes Paar von Transaktionen T_i, T_j $i \neq j$ aus AP, für das gilt: Zwischen T_i und T_j sind in AP Konfliktoperationen $x_i \rightarrow y_j$ spezifiziert.

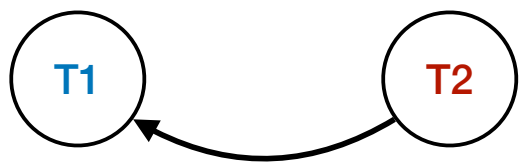
Die Leseweise der gerichteten Kante zwischen Transaktionen im Konflikt-Graphen **ist eine aggregierte Sicht der Konfliktoperationen** im Ausführungsplan (... `GROUP BY Transactions`...).

Beispiel: (Für unser Alice und Bob-Szenario von oben)

Aus dem Ausführungsplan...



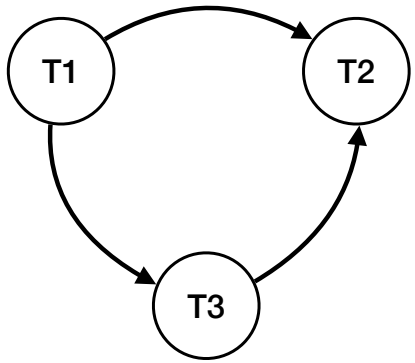
...können wir direkt den Konflikt-Graphen ablesen:



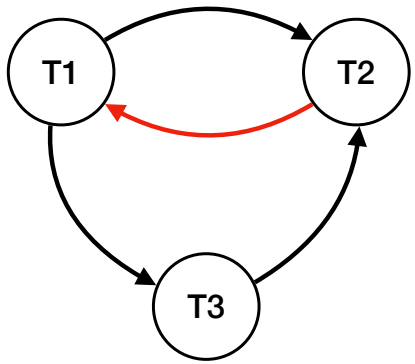
Beispiele

Hier haben wir die Commits weggelassen, da Sie für den Konflikt-Graphen keine Rolle spielen.

AP1: $r_1(A) \rightarrow r_3(B) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow r_3(A) \rightarrow w_2(A)$



AP2: $r_1(A) \rightarrow r_3(B) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow r_3(A) \rightarrow r_1(B) \rightarrow w_2(A)$



Warum entsteht die zusätzliche rote Kante im Konflikt-Graphen?

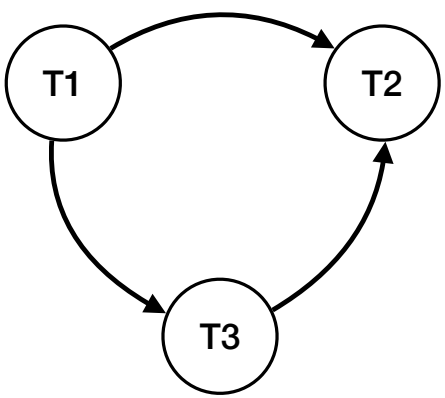
Weil es im Ausführungsplan AP2 die Konfliktoperationen $w_2(B)$ und $r_1(B)$ gibt.

Zyklus

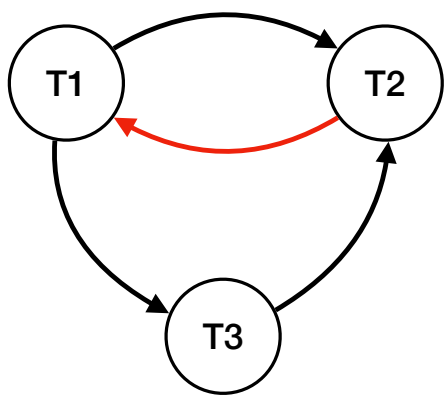
Zyklus

Mit Zyklus bezeichnen wir im Konflikt-Graphen jeden Pfad $T_i \rightarrow T_{j \neq i} \rightarrow \ldots \rightarrow T_i$.
Mit anderen Worten: wenn wir die Traversierung des Graphen bei T_i starten, wird der Knoten T_i durch die Traversierung erreicht (hierbei traversieren wir mindestens einen anderen Knoten). Der Graph ist somit zyklisch und **kein DAG** (directed acyclic graph).

Beispiel:



zyklenfreier Graph (DAG)



Graph mit Zyklus (kein DAG)

Konfliktserialisierbarkeit im Konflikt-Graphen

Konfliktserialisierbarkeit im Konflikt-Graphen

Ein Ausführungsplan ist konfliktserialisierbar genau dann, wenn der zugehörige Konflikt-Graph zyklenfrei ist.

Hieraus folgt direkt ein wichtiger (und praktikabler) Algorithmus, um einen nicht zyklenfreien Konflikt-Graphen konfliktserialisierbar zu machen:

Erstellen eines zyklenfreien Konflikt-Graphen

Ein nicht-zyklenfreier Konflikt-Graph lässt sich in einen zyklenfreien Konflikt-Graphen überführen, indem alle Transaktionen entfernt werden, die zu Zyklen führen.

Ausführungsplan vs physischer Plan

Ausführungsplan vs physischer Plan

Bei der Anfrageoptimierung hatten wir physische Pläne behandelt. Diese werden manchmal auch als *Ausführungspläne* bezeichnet. Bitte nicht verwechseln:

- **Ausführungsplan/Physischer Plan** (im Sinne von Anfrageoptimierung): mit Hilfe regel- und kostenbasierter Optimierung erstellter Ablaufplan zur Berechnung des Ergebnisses einer SQL-Anfrage oder eines Ausdrucks der relationalen Algebra
- **Ausführungsplan** (im Sinne von Serialisierbarkeit): Festlegung der Reihenfolge zwischen verschiedenen SQL-Anfragen/Änderungsoperationen (und somit physischen Plänen)

Batch vs Operation-at-a-time

Batch vs Operation-at-a-time

1. **Batch**: Eine Menge von Transaktionen soll ausgeführt werden. Wir können obige Graphanalyse ausführen, um einen serialisierbaren Ausführungsplan hierfür zu erzeugen. Z.B. mit Hilfe von topologischem Sortieren.
2. **Operation-at-a-time**: Eine Menge von Transaktionen sendet in beliebiger Reihenfolge Operationen zum Datenbanksystem. Wie können wir sicherstellen, dass diese nebenläufigen Operationen sich so verhalten wie ein serialisierbarer Ausführungsplan — und zwar **ohne abzuwarten, bis alle Transaktionen ihr commit geschickt haben!**

Beispiel zu Batch in:

[Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, Jens Dittrich: Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. SIGMOD 2019.]

Operation-at-a-time Algorithmen

Operation-at-a-time Algorithmen

Meistens wird bei der Verarbeitung von Transaktionen *Operation-at-a-time* angenommen. Hierfür brauchen wir einen Algorithmus, der garantiert, dass nebenläufige Transaktionen sich so verhalten wie ein serialisierbarer Ausführungsplan.

- Es gibt zahlreiche *Operation-at-a-time* Algorithmen.
- Im Folgenden zeigen wir den (historisch) Wichtigsten: *2PL (Two-Phase Locking)*.
- Moderne DBMS benutzen allerdings *MVCC (multi-version concurrency control)*.
- MVCC kann mit 2PL kombiniert werden, siehe Stammvorlesung

Sperren von Tupeln (Locking)

Sperren von Tupeln (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.

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

Two-Phase-Locking (2PL)

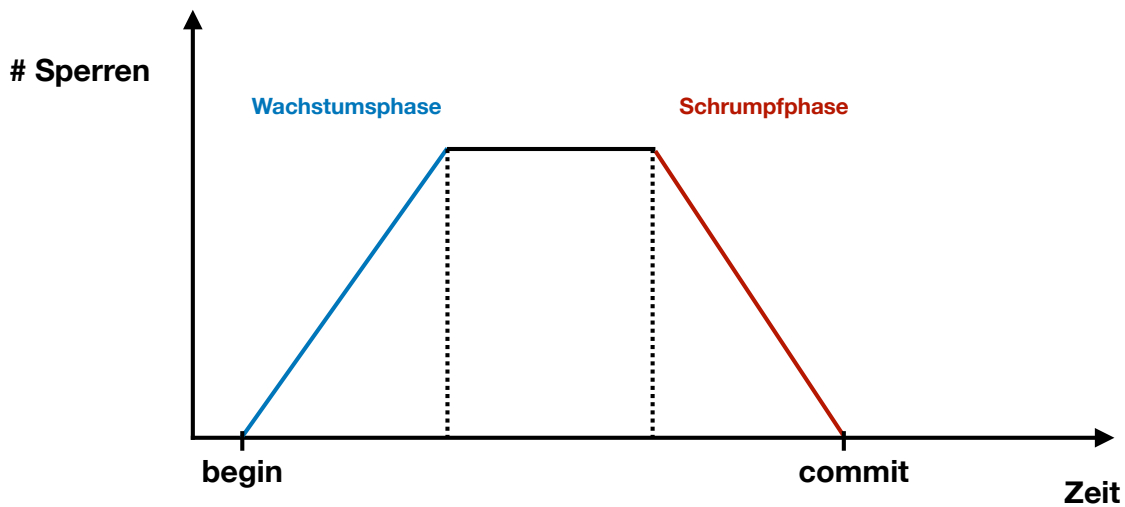
Two-Phase-Locking (2PL)

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:



2PL vs Konfliktserialisierbarkeit

2PL vs Konfliktserialisierbarkeit

2PL lässt nur Ausführungspläne zu, die konfliktserialisierbar sind.

OK, dann ist alles gut, oder?

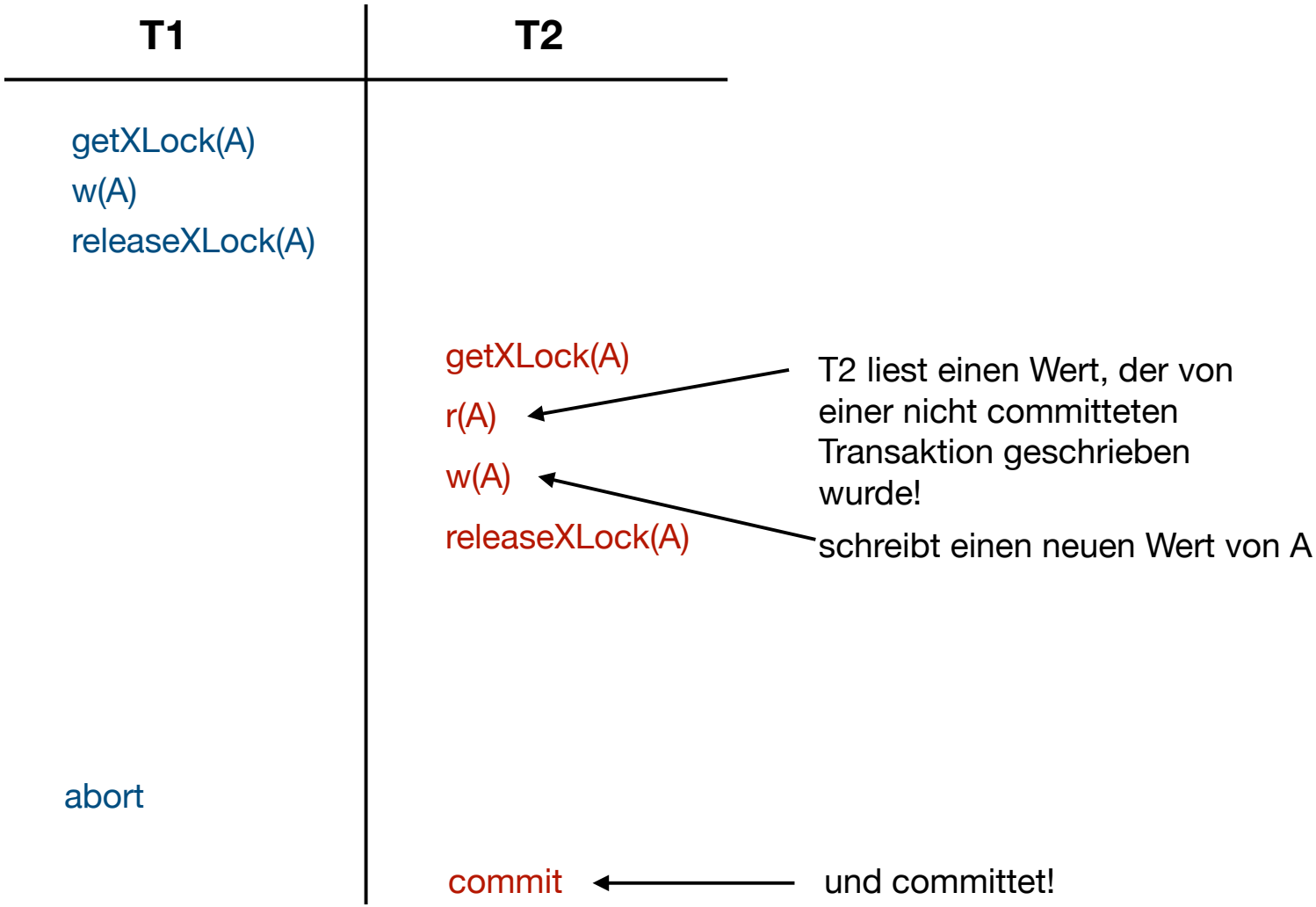
Leider nein, denn ein Ausführungsplan kann auch abgebrochene Transaktionen enthalten! Mit anderen Worten: jede Transaktion kann jederzeit vom Nutzer/Anwender abgebrochen werden.

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!

Kaskadierendes Zurücksetzen (cascading rollback)

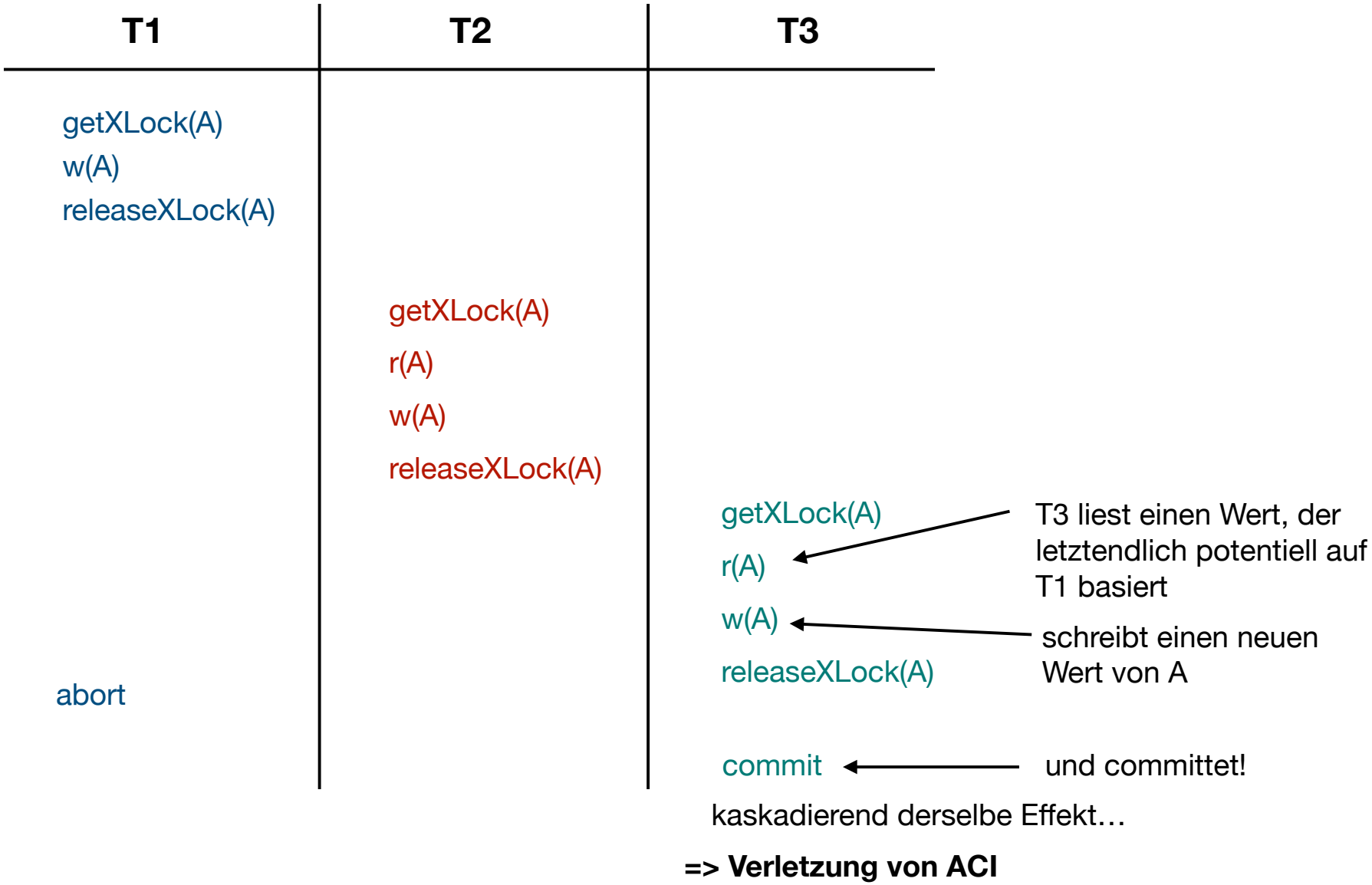
Beispiel: Dieser Ausführungsplan wird mit 2PL zugelassen (und ist konfliktserialisierbar)!



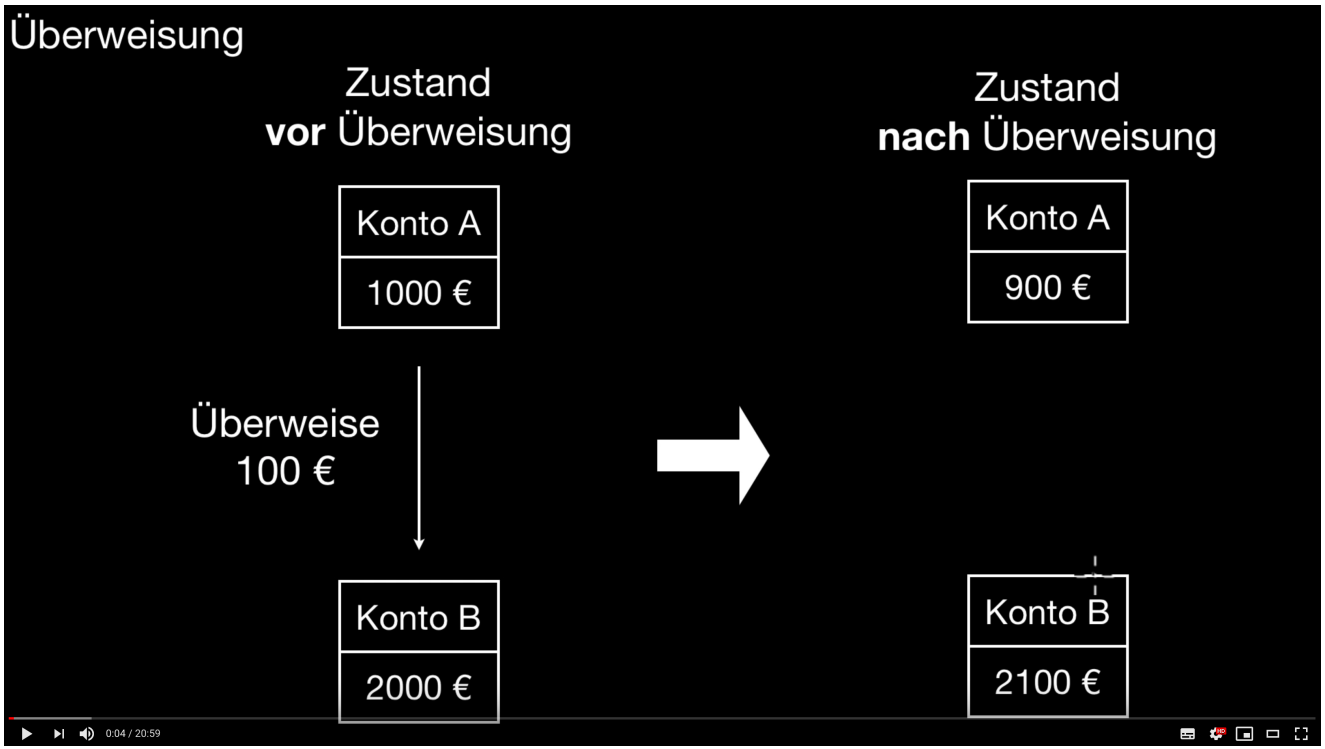
=> Verletzung von ACI

Kaskadierendes Zurücksetzen (cascading rollback)

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



Weiterführendes Material



- Video “Transaktionen und ACID”
- Kapitel in Kemper und Eickler