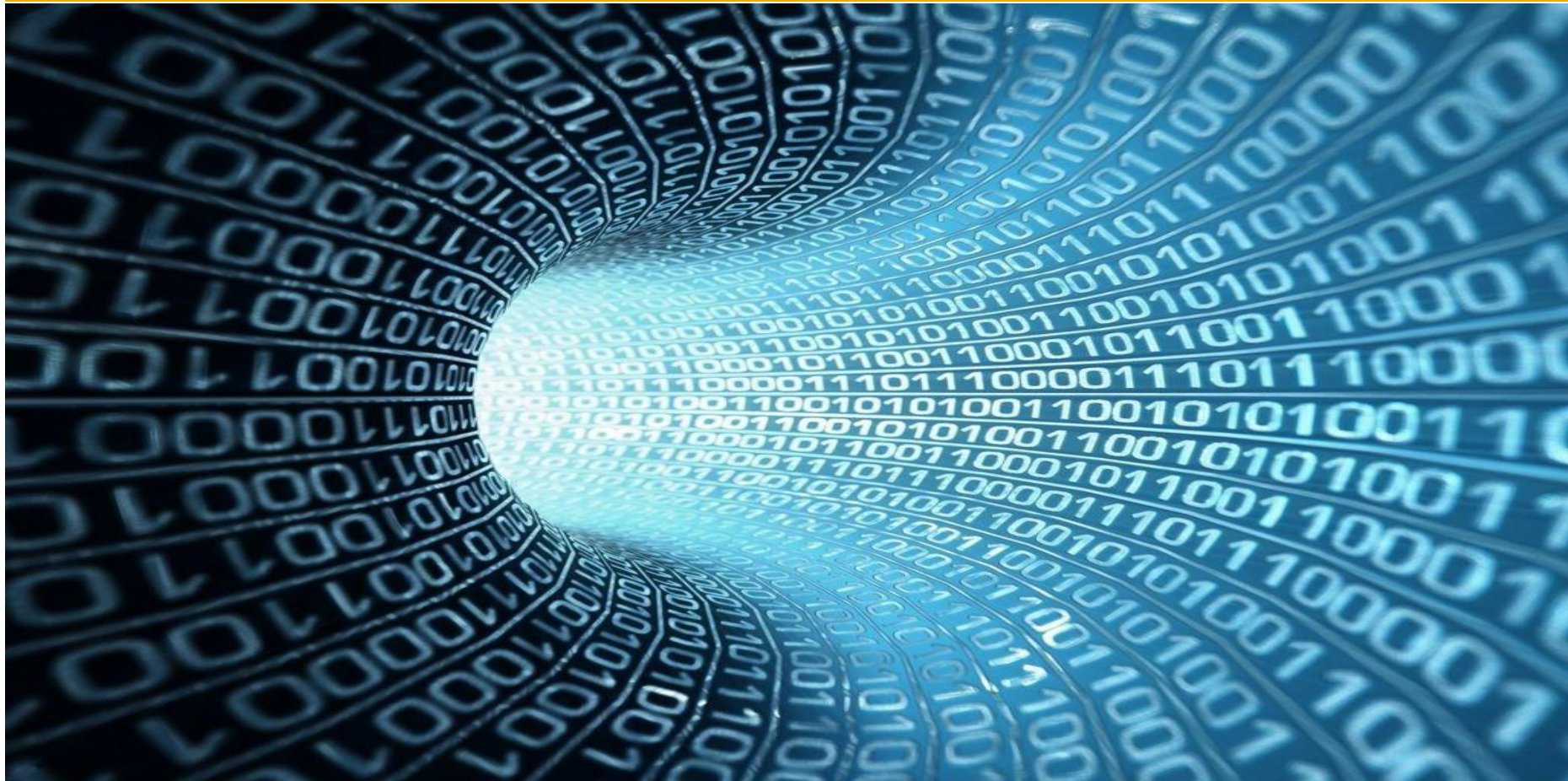


Vorlesung Datenbanken 2

Teil II: Transaktionsmanagement in relationalen Datenbanken

Prof. Dr. Zoltán Nocht



1. Einleitung

Allgemeiner Transaktionsbegriff

Transaktion (TA): Atomare Abfolge von Programmschritten, d.h. lesenden und schreibenden **Elementaroperationen an Nutzdaten** sowie **Systembefehlen**.

Es gibt genau zwei mögliche Abschlüsse einer Transaktion:

1. **Erfolgreicher** Abschluss nach/ durch **commit**
2. **Abbruch** (engl. **abort**) der Transaktion ansonsten

T_x : BOT, op_1 , op_2 , ..., op_i , ... op_m , **commit**, EOT

T_x : BOT, op_1 , op_2 , ..., **op_i** , **abort (rollback)**



- **Commit:** DBMS schreibt alle Änderungen im persistenten Datenbestand (Dateien auf Hintergrundspeicher) fest.
- **Abort:** Alle Änderungen der TA werden vom DBMS **rückgängig** gemacht (engl. *rollback*), um Konsistenz der Datenbank sicherzustellen.

Transaktionen

Beispiele für Systembefehle

- **begin/end of transaction (BOT, EOT):** Beginn/Ende der Transaktion.
- **commit:** Leitet die Persistierung der neuen/aktualisierten Daten ein. Zu Beginn des Vorgangs wird 'Commit' im Logfile eingetragen.
- **abort:** Führt zum Abbruch der Transaktion. DBMS leitet das Zurücksetzen **aller** durch TA durchgeführter Änderungen ein. Der Abbruch einer TA kann zu einer **Abbruch-Kaskade** mehrerer TA führen. Beachte: Das DBMS kann eine TA auch *ohne* einen abort-Befehl selbständig abbrechen, bspw. um **Verklemmungen** (engl. *deadlock*) aufzulösen.
- **define savepoint:** Definiert einen **Sicherungspunkt**, auf den sich die noch aktive TA ggfs. zurücksetzen lässt. DBMS merkt sich dazu alle bis zu diesem Zeitpunkt ausgeführten Änderungen der Daten. Sie dürfen aber noch nicht persistiert werden, da die TA durch ein *abort* immer noch gänzlich aufgegeben werden kann.
- **backup transaction:** Zurücksetzen der noch aktiven TA typischerweise auf den zuletzt angelegten (d.h. jüngsten) Sicherungspunkt.

Transaktion als Pseudocode

Beispiel #1

- **Beispiel:** Umbuchung von Giro- auf Sparkonto in abstrakter Form:
 1. Lese den Kontostand von Girokonto A in Variable a :
read(A, a);
 2. Reduziere den Kontostand um 50 Euro:
 $a := a - 50$;
 3. Schreibe den neuen Kontostand von A in die Datenbasis:
write(A, a);
 4. Lese den Kontostand von Sparkonto B in die Variable b :
read(B, b);
 5. Erhöhe den Kontostand von B um 50 Euro:
 $b := b + 50$;
 6. Schreibe den neuen Kontostand in die Datenbasis:
write(B, b);
 7. **Commit;**

Beachte: Die *implizite Konsistenzbedingung* im Beispiel ist, dass die Umbuchung weder Kunden noch die Bank reicher/ärmer macht...

Transaktion in SQL

Beispiel #2

■ **Beispiel:** Umbuchung von Giro- auf Sparkonto in SQL-Syntax:

--Dieser Befehl verhindert die frühzeitige Persistierung einzelner SQL-Anfragen

AUTOCOMMIT FALSE 

BEGIN TRANSACTION

-- Auswahl und Aenderung einzelner Konten eines Kunden

UPDATE T_Giro SET Deposit=Deposit-50 WHERE KundenNr = 42;

UPDATE T_Spar SET Deposit=Deposit+50 WHERE KundenNr = 42;

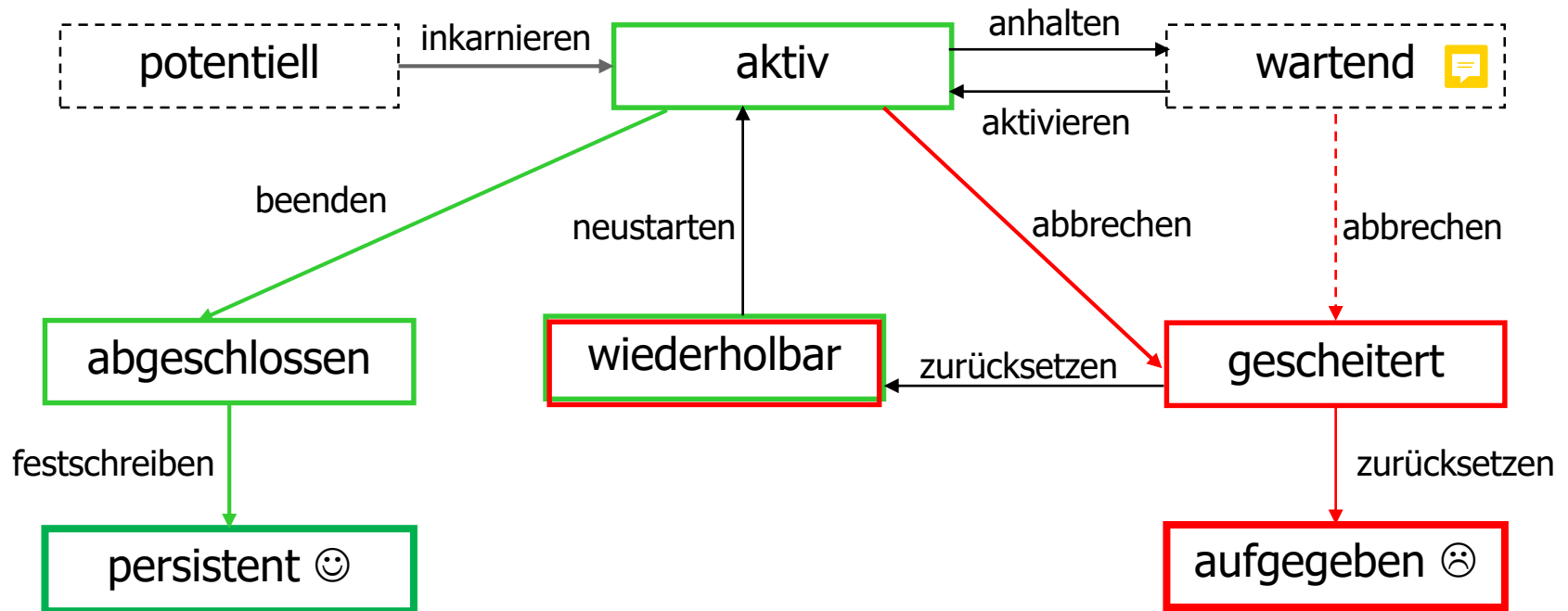
COMMIT

END

Ausführung von Transaktionen

Zustandsübergangs-Diagramm

- DBMS implementiert und überwacht die korrekte Durchführung einzelner Transaktionen gemäß einem Zustandsdiagramm, bspw.:



- Das Zustandsdiagramm hängt mit dem Synchronisationsverfahren des DBMS eng zusammen (Sperrren vs. Snapshots).

Zur Erinnerung: Das ACID-Paradigma

ACID ist ein **Leitprinzip** für Konstrukteure und Betreiber von DBMS:

■ Atomicity – Atomarität:

- „Alles oder nichts“ – Entweder alle oder gar keine Änderungen einer Transaktion werden in der Datenbasis festgeschrieben.

■ Consistency – Integrität:

- Jede erfolgreiche Transaktion führt die Datenbank von einem konsistenten in einen anderen konsistenten Zustand über.
- Entsprechend den definierten Integritätsbedingungen

■ Isolation – Isolation:

- Jede Transaktion hat die Datenbank ‚für sich allein‘, d.h. parallel laufende Transaktionen beeinflussen sich gegenseitig nicht.

■ Durability – Dauerhaftigkeit:

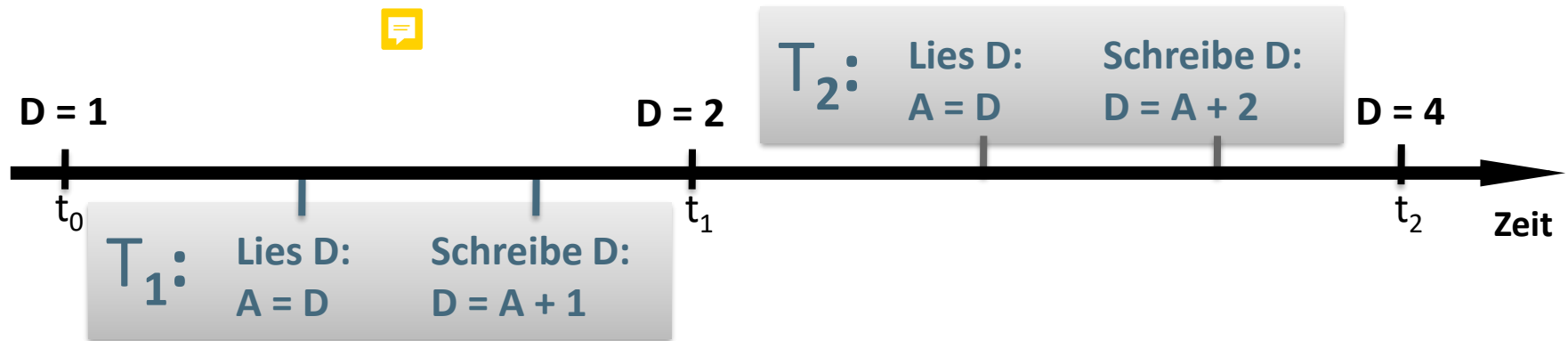
- Änderungen erfolgreicher Transaktionen (und nur die!) dürfen nicht verloren gehen.

2. Synchronisation von Transaktionen

Serielle Durchführung von Transaktionen

Transaktionen, die teilweise auf gleiche Datenbankobjekte abzielen, können **zeitlich nacheinander**, d.h. **seriell**, ablaufen.

- Das kann entweder **Zufall** sein, oder
- DBMS **startet** T_2 erst nach Beendigung von T_1 :




- Jede serielle Durchführung von T_i Transaktionen in einer beliebigen Reihenfolge (!) führt die Datenbank von einem anfangs konsistenten Zustand in einen neuen konsistenten Zustand über.
- Folglich kann es mehrere konsistente Zustände zum Zeitpunkt t geben. Wenn im Beispiel T_2 vor T_1 läuft, ist der Zustand $D=3$ zu t_1 konsistent.

Konsistenz im Transaktionsbetrieb

- **Konsistenz:** Änderungen, die eine Transaktion tätigt, erfüllen alle in der DB **explizit definierten Konsistenzbedingungen**, zum Beispiel:
 - Werte der Spalte *Gehalt* in *mitarbeiter* müssen mindestens 0 sein.
 - Summe der Gehälter darf nicht 2.000.000 überschreiten.
 - Werte in *PNrSales* in *absatz* dürfen nur Werte von *PNr* in *mitarbeiter* aufnehmen, wobei für den Mitarbeiter *Job*='Sales' gilt.
- Andere, dem DBMS unbekannte Regeln, kann es von sich aus nicht erfüllen. Zum Beispiel:
 - In der Bibliothek-App muss die Transaktion „*Buch_verleihen*“ immer **vor** der Transaktion „*Buch_zurückgeben*“ durchgeführt werden.
 - In der Reise-App muss nach „*Flug_stornieren*“ immer auch „*Hotel_stornieren*“ erfolgen, nicht aber umgekehrt.

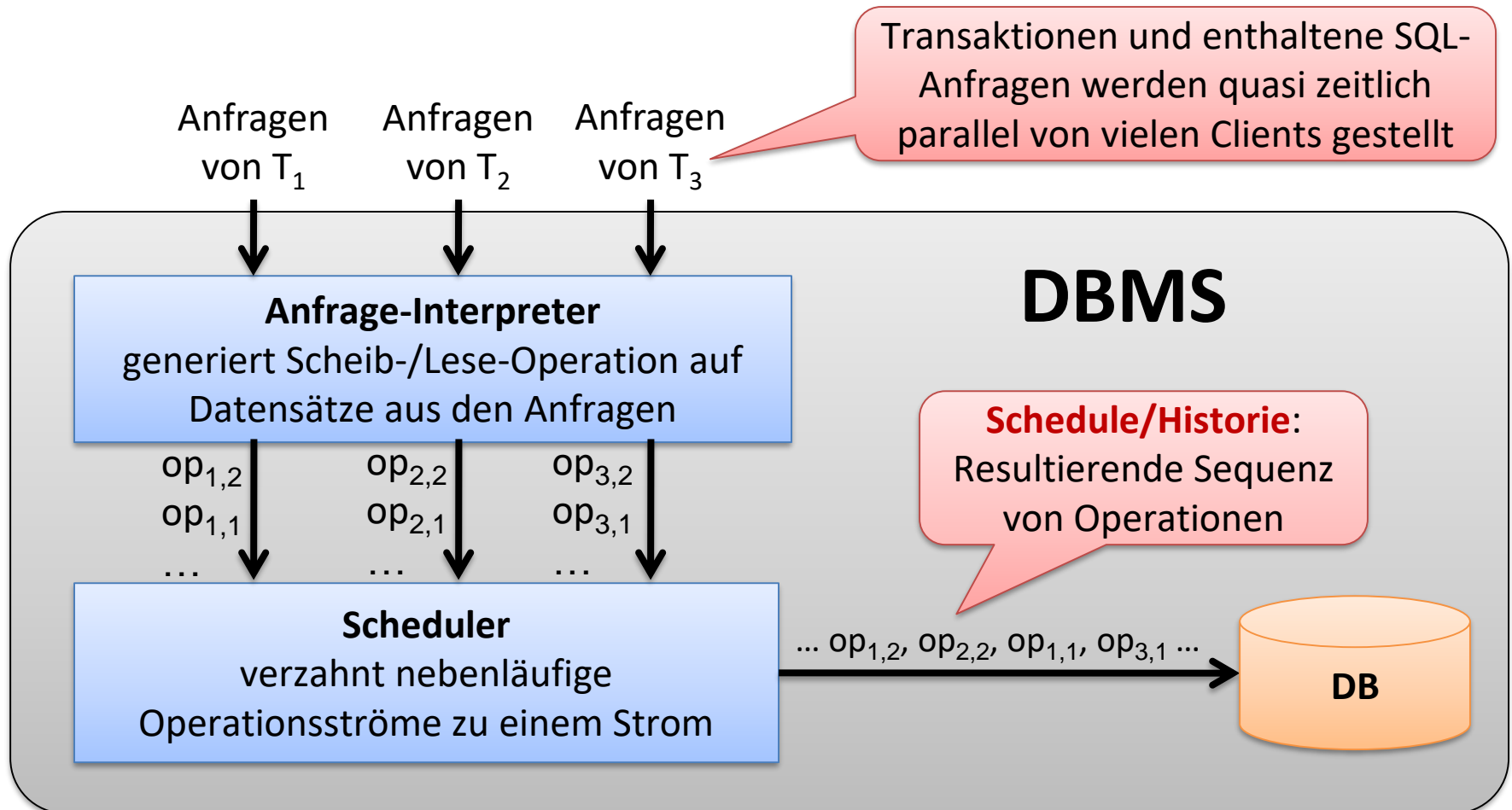
Durchführung von nebenläufigen Transaktionen

Isolation und Konflikte

- Die serielle Durchführung nutzt Systemressourcen **suboptimal** aus:
 - Wegen langsamer I/O-Zugriffe in T_1 muss die CPU unnötig lange warten, bevor er mit Verarbeitung von T_2 beginnen kann.
- Deshalb werden Transaktionen in der Praxis  **verzahnt** durchgeführt.
- **Scheduler** „verzahnt“ Operationen einzelner TA (engl. *Interleaving*).
- **Isolation:** Der Gesamteffekt der Verzahnung ist trotzdem so, als ob die Transaktionen nicht parallel, sondern seriell (also streng nacheinander und folglich korrekt) ablaufen würden.
- **Konflikte** entstehen, wenn zwei oder mehr Transaktionen auf dasselbe DB-Objekt D abzielen und mindestens einer der TA D schreiben (verändern, löschen) möchte.

Verzahnung von Transaktionen

- Scheduler führt einzelne Lese- und Schreibe-Operationen voneinander isolierter parallel laufender Transaktionen in der Datenbank aus.



Darstellung von Schedules (Historien)

- Ein **formalisierter Schedule** (auch als **Historie** bezeichnet) ist eine Folge von elementaren Operationen:
 - $r_i(x)$: Lesen des Datensatzes x durch die Transaktion T_i
 - $w_i(y)$: Schreiben des Datensatzes y durch T_i
 - c_i : Commit, d.h. erfolgreicher Abschluss von T_i
 - a_i : Abort, d.h. nicht erfolgreicher Abschluss von T_i
- Von eigentlichen Werten, Größe, oder Struktur der Datensätze x, y, z , etc. wird abstrahiert. Auch werden etwaige Abhängigkeiten zwischen x, y, z, \dots ignoriert.

Diese Eigenschaften sind aus Sicht des Schedulers *uninteressant*, da dieser eine Historie **nur** im Sinne der angestrebten Isolation bzw. Serialisierbarkeit untersucht.

Äquivalenz und Serialisierbarkeit von Transaktionen

- Beispiel für einen seriellen Schedule S_1 : $\mathbf{T}_1 \mid T_2 \mid T_3$

$$S_1 = \mathbf{r_1(z)} \mathbf{w_1(x)} \mathbf{w_1(z)} \mathbf{c_1} \mathbf{r_2(x)} \mathbf{r_2(y)} \mathbf{c_2} \mathbf{w_3(y)} \mathbf{r_3(z)} \mathbf{c_3}$$

- Ein anderer, nicht serieller Schedule S_2 mit denselben Operationen:

$$S_2 = \mathbf{r_1(z)} \mathbf{r_2(x)} \mathbf{w_3(y)} \mathbf{r_2(y)} \mathbf{w_1(x)} \mathbf{r_3(z)} \mathbf{c_3} \mathbf{c_2} \mathbf{w_1(z)} \mathbf{c_1}$$

Äquivalenz und Serialisierbarkeit von zwei Schedules S_i und S_j :

- Wenn S_j den gleichen Zielzustand auf einer DB erzeugt wie S_i , dann sind S_i und S_j *äquivalent*.
- Wenn S_i seriell ist, dann ist der mit S_i äquivalente S_j *serialisierbar*.
- Ein serieller Schedule ist auch immer serialisierbar, aber nicht jeder serialisierbare Schedule ist seriell...

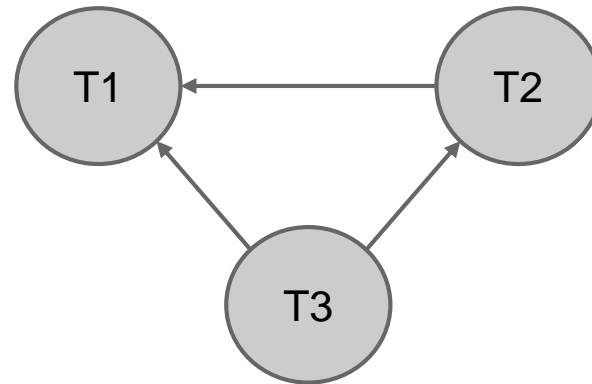
Serialisierbarkeit von Transaktionen

- Ist der Schedule S_2 serialisierbar? Gibt es mind. einen seriellen (und folglich konsistenten) Schedule S' , der mit S_2 äquivalent ist?

$S_2 = r_1(\textcolor{red}{z}) \ r_2(\textcolor{red}{x}) \ w_3(y) \ r_2(y) \ w_1(\textcolor{red}{x}) \ r_3(\textcolor{red}{z}) \ c_3 \ c_2 \ w_1(\textcolor{red}{z}) \ c_1$ 🗨️

Prüfung der Serialisierbarkeit von S_2 mit Serialisierungsgraphen $SG(S_2)$:

- T3 vor T2 (wegen y)
- T2 vor T1 (wegen x)
- T3 vor T1 (wegen $\textcolor{red}{z}$)



- $SG(S)$ kreisfrei \leftrightarrow S serialisierbar.
- Prüfung der Serialisierbarkeit ist in DBMS unter Last i.d.R. nicht praktikabel. Statt dessen werden Scheduler benötigt, die Konflikte in den erzeugten Schedules ausschließen.

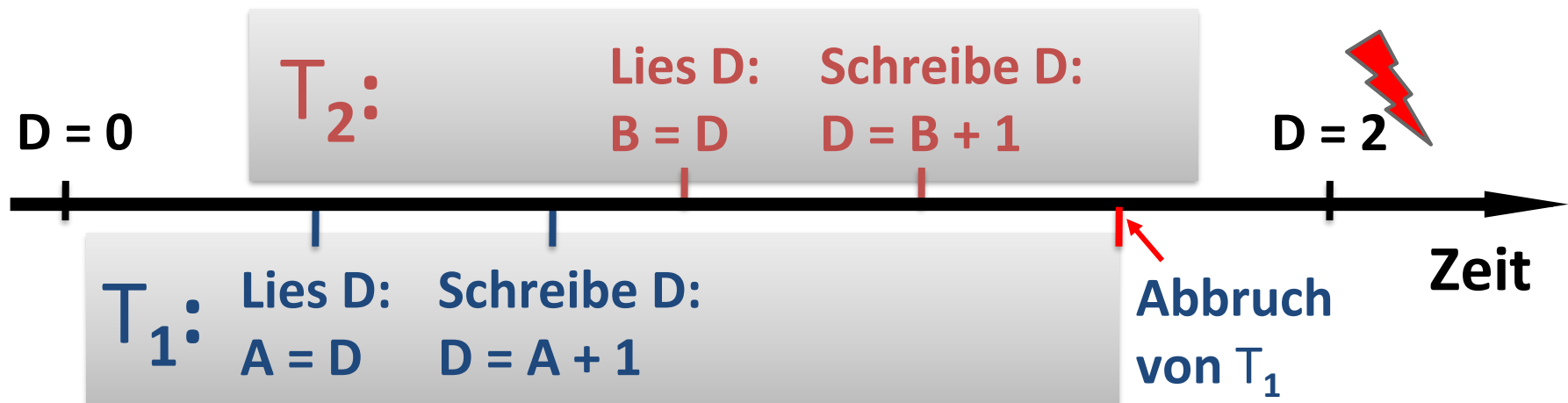
Inkonsistenz-Phänomene durch fehlerhafte Verzahnung

Dirty Read

Dirty Read: T_2 liest den von T_1 noch nicht festgeschriebenen Datensatz D .

Warum ist es ein Problem?: 

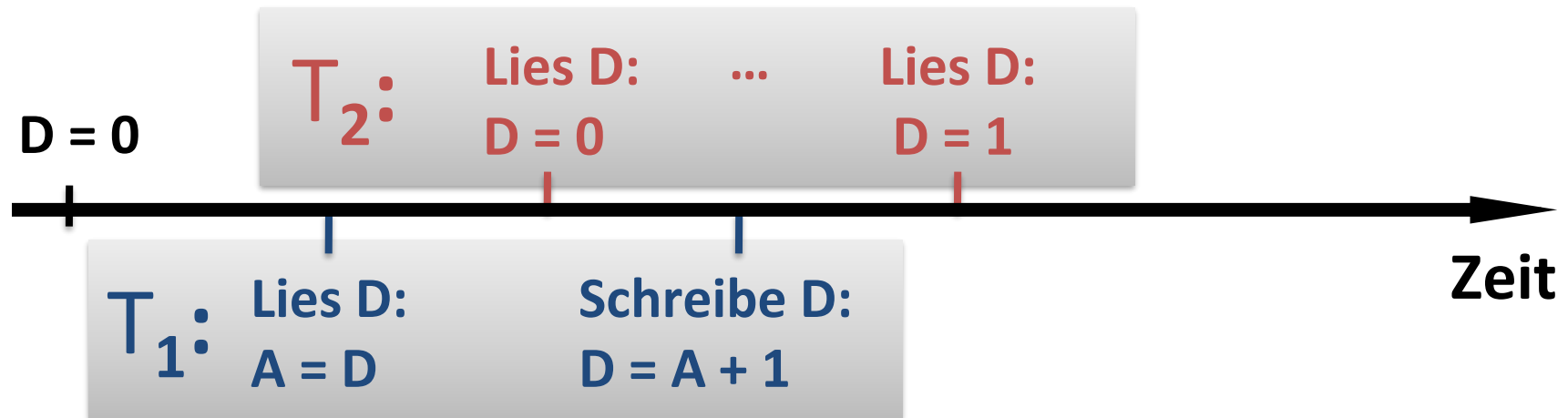
- Wenn T_1 abstürzt, ist das Zurücksetzen von T_1 *und* T_2 notwendig, damit D den vorher korrekten Wert behält.
- Im Beispiel muss D also wieder auf $D=0$ zurückgesetzt werden.



Inkonsistenz-Phänomene durch fehlerhafte Verzahnung

Non-Repeatable Read

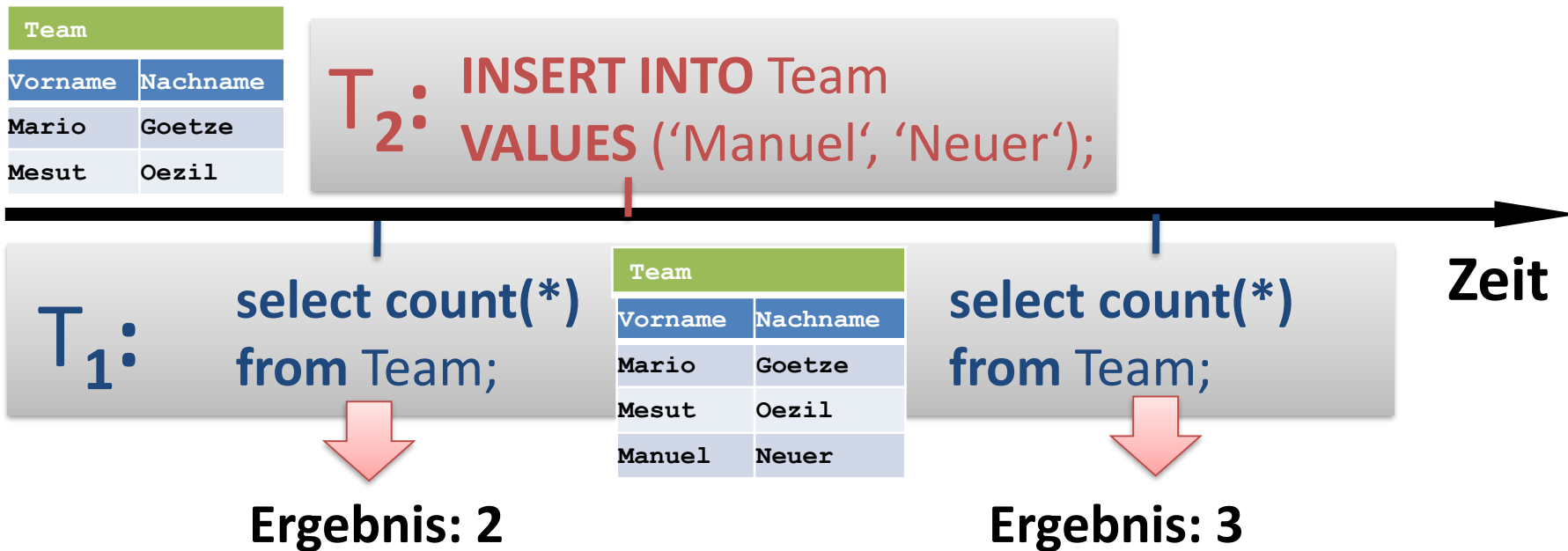
- T_2 liest zweimal den Wert von D :
 - beim ersten Lesevorgang ist $D = 0$,
 - beim zweiten ist $D = 1$, da T_1 den Datensatz inzwischen überschrieb.



Inkonsistenz-Phänomene durch fehlerhafte Verzahnung Phantome

Phantom:

- T_1 listet/zählt Zeilen einer Tabelle,
- T_2 fügt eine neue Zeile ein,
- T_1 listet/zählt nochmals und erhält einen neuen Wert



Inkonsistenz-Phänomene

Anmerkung zum Phantomproblem

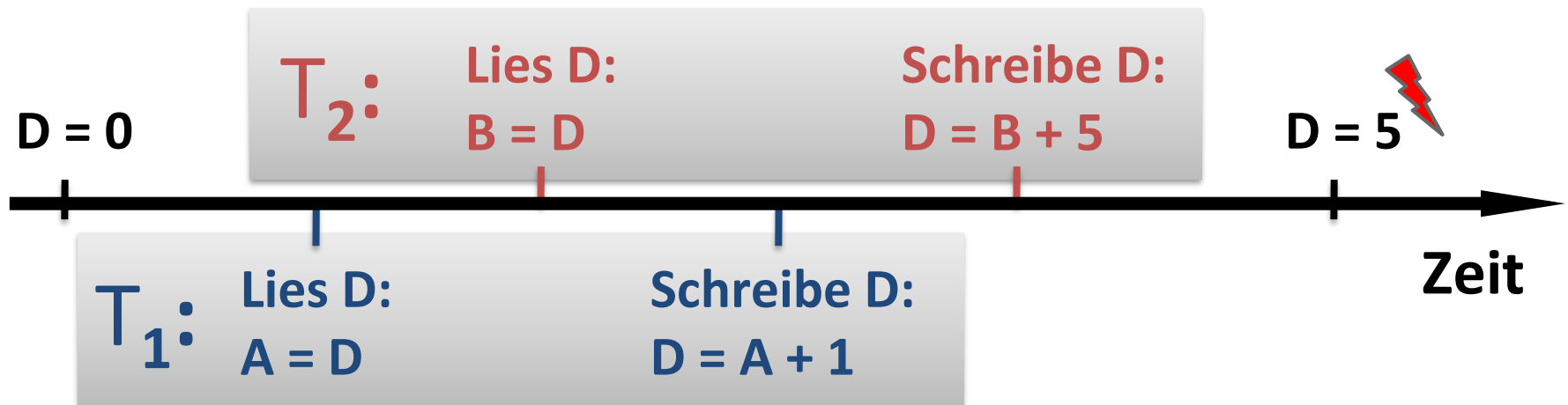
- Beachte: Das Phantomproblem ist im Wesentlichen das gleiche Phänomen wie das **Non-Repeatable Read Problem**
- Allerdings betrifft Non-Repeatable Read nur einen Datensatz – das **Phantomproblem eine bestimmte Menge von Datensätzen**, wie bspw. ganze Tabelle, oder Wertebereiche (gemäß WHERE-Klausel)
- Wenn man also größere Einheiten als Datensätze (z.B. Tabellen) betrachtet, dann sind die beiden Phänomene identisch

Inkonsistenz-Phänomene durch fehlerhafte Verzahnung

Lost Update

Lost Update: Die Änderung von T_1 geht (für T_2) **verloren**:

- Am Ende von T_2 müsste D den Wert $D=6$ anstatt $D=5$ haben.
- Wenn T_2 den neuen Wert von D von T_1 liest, ist es ein Dirty Read...



Inkonsistenz-Phänomene im Überblick

- **Dirty Read:** Eine Transaktion liest einen Datensatz, der von einer später abgebrochenen Transaktion geschrieben wurde.
- **Non-repeatable Read:** Eine Transaktion liest zweimal den gleichen Datensatz, zwischendurch wird der Datensatz aber von einer anderen Transaktion verändert.
- **Phantomproblem:** Eine Transaktion T_2 verändert/erzeugt Datensätze innerhalb einer Datenmenge, die vorher als Ganzes von T_1 gelesen wurde. Die Menge der gelesenen Datensätze erscheint T_1 bei einem zweiten Lesen verändert.
- **Lost Update:** Die Änderung eines Datensatzes durch eine Transaktion geht verloren, weil eine andere Transaktion denselben Datensatz zwischenzeitlich überschreibt.

Transaktionen vor Fehlerphänomenen schützen

- Im SQL-Standard gibt es **vier** Stufen zur Vermeidung von Fehlerphänomenen:

Level-Name in SQL	Dirty Read	Non-Repeatable Read	Phantom Problem
READ UNCOMMITTED	möglich	möglich	möglich
READ COMMITTED	nein	möglich	möglich
REPEATABLE READ	nein	nein	möglich
SERIALIZABLE	nein	nein	nein

- Leider ist die Definition der Stufen unscharf, und es gibt weitere, im Standard nicht beachtete Phänomene.

Mehr dazu in *Berenson et al: „A Critique of ANSI SQL Isolation Levels“*

<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf>

- Ferner war eine implizite Annahme bei der Festlegung der Stufen, dass die Isolation durch **Sperren** erfolgt.
- Aus diesen Gründen führten DBMS-Hersteller weitere **proprietäre Stufen** ein, wie z.B. *Cursor Stability* in IBM DB2, *Snapshot Isolation* in Oracle, PostgreSQL,...
 - Dies erschwert das Testen und die Portierbarkeit von Anwendungen.

Hinweise zu Isolation Levels

- Die meisten DBMS-Produkte haben eine voreingestellte Stufe, die innerhalb einer DBMS-Instanz normalerweise gilt, bspw. READ COMMITTED in PostgreSQL, MS SQL Server 2008, Oracle.
- Eine Abweichung davon ist meistens für die Verarbeitung einzelner Transaktionen eines mit dem DBMS verbundenen Clients (für die Dauer der Session!) möglich, z.B. durch

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE (Oracle)

SET TRANSACTION ISOLATION LEVEL SNAPSHOT (MS SQL Server)

- „Interessante“ Situationen können sich ergeben, wenn Transaktionen verschiedener Clients unter verschiedenen Isolationsstufen laufen (dürfen) und auf dieselben DB-Objekte zugreifen...

Sperren werden im DBMS durch den **Lock Manager** (Sperrverwalter) vergeben. Es gibt zwei grundlegende Sperrmodi:

- **Lesesperre S** (engl. **Shared lock**, oder read lock):
 - $S(A)$: Lesesperre auf Datensatz **A**
 - $LockS(A)$: Anforderung der Lesesperre beim Lock Manager
 - Wenn eine TA Lesesperre $S(A)$ besitzt, kann sie $read(A)$ ausführen
 - Mehrere Transaktionen können gleichzeitig eine Lesesperre für denselben Datensatz besitzen und diesen lesen.
- **Schreibsperre X** (engl. **Exclusive lock**, oder write lock):
 - $X(A)$, $LockX(A)$
 - Nur die Transaktion, die $X(A)$ besitzt, darf $write(A)$ ausführen.
- Gesperrt werden i.d.R. physische Datenbankobjekte, d.h. **Datensätze** (Zeile), **Seiten** (Tabelle, Index), **Dateien** (Datenbank).

Verträglichkeitsmatrix für Sperren

Welche Transaktion darf eine **für ein bestimmtes Datenobjekt D** beantragte Sperre bekommen?

- **Verträglichkeitsmatrix** für Sperren (auch **Kompatibilitätsmatrix** genannt):

	<i>NL</i>	<i>S</i>	<i>X</i>
<i>S</i>	✓	✓	-
<i>X</i>	✓	-	-

Wie liest man die Matrix?

- Wenn D nicht gesperrt ist (NL), können sowohl S- wie X-Sperren gewährt werden.
- Wenn S-Sperre gesetzt wurde, darf eine S- aber keine X-Sperre gesetzt werden. (Ausnahme: Upgrade einer erhaltenen S-Sperre durch dieselbe Transaktion möglich, wenn keine weiteren S-Sperren vergeben sind.)
- Wenn eine X-Sperre vergeben ist, darf keine weitere Sperrung erfolgen.

Serialisierung und Sperrverfahren

Two Phase Locking (2PL)

Das **Zwei-Phasen Sperrprotokoll** (engl. *Two-Phase Locking*, **2PL**) ist ein weitverbreiteter Ansatz zur Verzahnung nebenläufiger Transaktionen.

Das sind die **2PL-Regeln**:

- Jedes Objekt, das von einer TA benutzt werden soll, muss vorher gesperrt werden. Eine TA fordert eine Sperre, die sie schon besitzt, nicht erneut an.
- Jede Transaktion durchläuft zwei Phasen:
 - **Wachstumsphase**, in der sie Sperren anfordern, aber **keine** freigeben darf
 - **Schrumpfungsphase**, in der sie ihre bisher erworbenen Sperren nacheinander freigibt, aber keine weiteren anfordern darf.
- Wenn eine Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende **Warteschlange** (s. auch Zustandsdiagramm!) eingereiht – bis die Sperre gewährt werden kann.
- Bei EOT (Transaktionsende) muss eine Transaktion **alle** ihrer evtl. noch vorhandenen Sperren zurückgeben.

Two Phase Locking

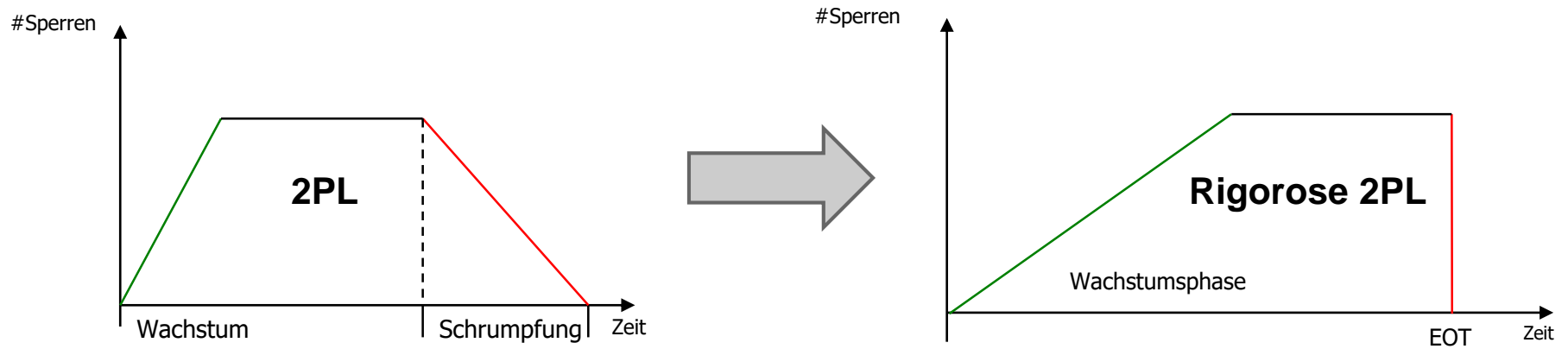
Beispiel

Beispiel für die Anwendung von 2PL bei zwei konkurrierenden Transaktionen:

#	T_1	T_2	Bemerkung
1.	BOT		
2.	lockX(A)		
3.	read(A)		
4.	write(A)		
5.		BOT	
6.		lockS(A)	T_2 muss warten
7.	lockX(B)		
8.	read(B)		
9.	unlockX(A)		T_2 wecken
10.		read(A)	(Dirty Read!)
11.		lockS(B)	T_2 muss warten
12.	write(B)		
13.	unlockX(B)		T_2 wecken
14.		read(B)	(Dirty Read!)
15.	commit		
16.		unlockS(A)	
17.		unlockS(B)	
18.		commit	

Rigorese Zwei-Phasen-Sperrprotokolle

- Im **Strict 2PL (S2PL)** Protokoll werden erhaltene **Schreibsperren** „gleichzeitig“ am Ende der Transaktion freigegeben.
- Im **Strong Strict 2PL (SS2PL)** werden **alle Lese- und Schreibsperren** erst zu Transaktionsende freigegeben.



- Diese Verschärfung bringt Vor- und Nachteile mit sich:
 - Vereinfachte Sperrverwaltung 😊
 - Transaktionen können länger blockiert werden ☹️

Regeln des **Strong Strict 2PL (SS2PL)**:

■ Für Leseoperation $r_i(x)$ gilt:

Wenn eine andere Transaktion T_j eine **Schreibsperre** auf x hält, wird $r_i(x)$ blockiert, bis T_j beendet ist.

Danach wird eine Lesesperre auf x gesetzt und $r_i(x)$ ausgeführt.

■ Für Schreiboperation $w_i(x)$ gilt:

Wenn eine andere Transaktion T_j eine **Lese- oder Schreibsperre** auf x hält, wird $w_i(x)$ blockiert, bis T_j beendet ist.

Danach wird eine Schreibsperre auf x gesetzt und $w_i(x)$ ausgeführt.

■ SS2PL **schließt** beim Scheitern einer Transaktion **Abbruchkaskaden aus**.

■ **Verteilte Transaktionen können ohne einen zentralen Lock-Manager** ausgeführt werden, wenn alle Teiltransaktionen an allen DMBS-Knoten gemäß SS2PL laufen.

SS2PL

Funktionsweise am Beispiel

- Der Schedule S_2 von vorhin:

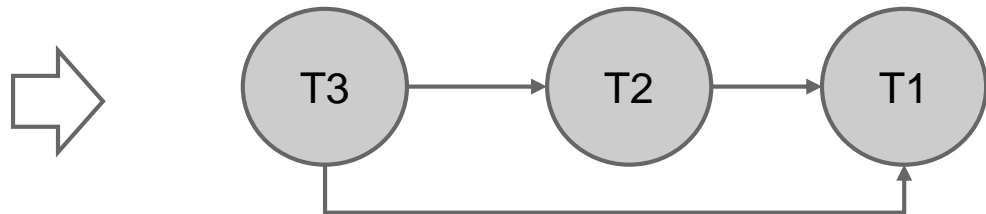
$$S_2 = r_1(z) \ r_2(x) \ w_3(y) \ \mathbf{r_2(y)} \ \mathbf{w_1(x)} \ r_3(z) \ c_3 \ c_2 \ w_1(z) \ c_1$$

- SS2PL erzeugt daraus durchs **Blockieren** von zwei Operationen diesen neuen Schedule:

$$S_{SS2PL} = r_1(z) \ r_2(x) \ w_3(y) \ r_3(z) \ c_3 \ \mathbf{r_2(y)} \ c_2 \ \mathbf{w_1(x)} \ w_1(z) \ c_1$$

- SS2PL garantiert die Serialisierbarkeit eines Schedules in der Commit-Reihenfolge der Transaktionen.
D.h. die serielle Abfolge $T_3 \mid T_2 \mid T_1$ ist mit S_{SS2PL} äquivalent.

- T3 vor T1 wg. z
- T3 vor T2 wg. y
- T2 vor T1 wg. x



Serialisierung und Sperrverfahren

Deadlocks durch Sperren

Wenn T_1 auf die Freigabe von Sperren von T_2 wartet, während T_2 wegen von T_1 gesetzten Sperren ebenfalls wartet, spricht man von einer **Verklemmung** (eng. *deadlock*).

■ Am Beispiel:

Schritt	T_1	T_2	Bemerkung
1.	BOT		
2.	lockX(A)		
3.		BOT	
4.		lockS(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	lockX(B)		T_1 muss warten auf T_2
9.		lockS(A)	T_2 muss warten auf T_1
10.	\Rightarrow <i>Deadlock</i>

■ Gegenmaßnahmen:

- Erkennen und aktives Auflösen durch Abbruch von Transaktionen (SS2PL)
- Vermeidung (z.B. durch Verzicht auf Sperren)

Synchronisation ohne Sperren

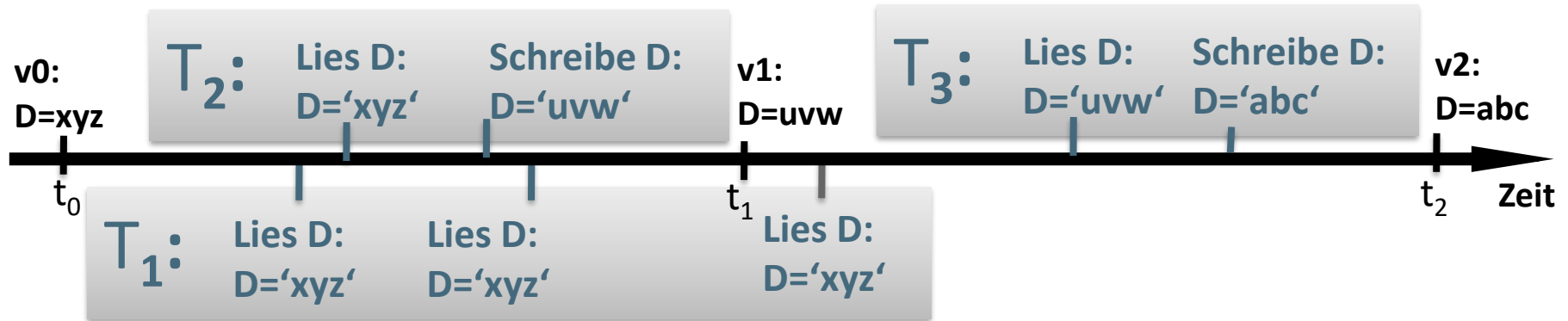
Multi-Version Concurrency Control (MVCC)

MVCC a.k.a. **Snapshot Isolation** ermöglicht die Synchronisation konkurrierender Zugriffe, **ohne** Datenbankobjekte zu sperren.

- Es wird in den meisten RDBMS und auch NoSQL-Systemen verwendet.
- MVCC begünstigt vor allem das **konkurrierende Lesen**:
 - Zu jedem Zeitpunkt steht **eine aktuelle Version** („Snapshot“) eines jeden Datensatzes für alle Transaktionen zum Lesen bereit.
 - Diese Version mag inzwischen **veraltet**, d.h. durch andere noch laufende Transaktionen mittlerweile überschrieben, sein.
 - Das Lesen ist vom Schreiben völlig **entkoppelt**, d.h. es gibt keine Wartezeiten für Leseoperationen.

MVCC

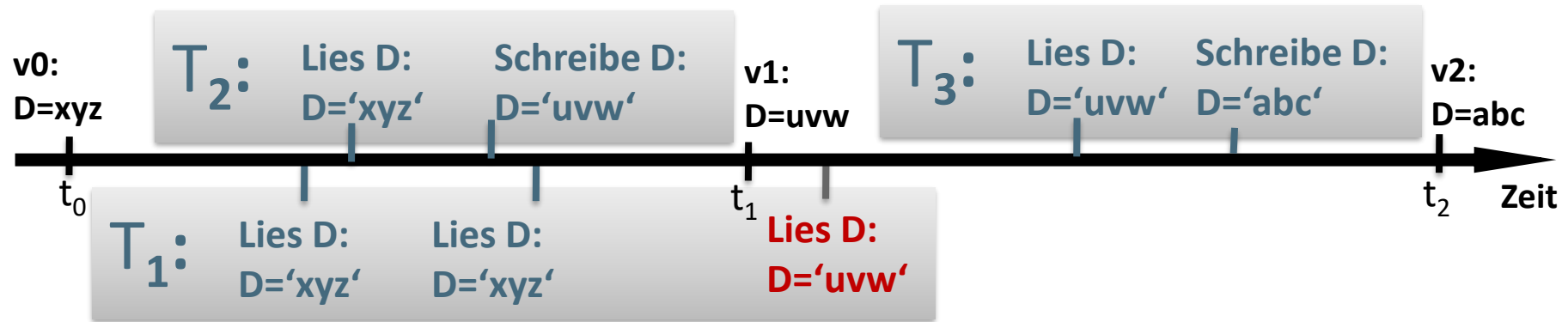
Erläuterung sperrfreies Lesen



- T_2 erzeugt ausgehend von $v0$ von D ('xyz') eine neue Version $v1$ ('uvw'). Nach Commit von T_2 ist $v1$ die neue aktuelle Version des Datensatzes.
- T_1 arbeitet mit der zum Startzeitpunkt gültigen Version $v0$. Dies gilt für alle Transaktionen, die *vor* t_1 gestartet sind (engl. **Transaction-level Snapshot**).
- T_3 liest und überschreibt $v1$ von D . Nach Commit ist $v2$ gültig ('abc').
- Solange der Datensatz ausgehend von einer alten Version nicht überschrieben werden soll, kommt es zu keinem Konflikt.

MVCC

Alternative Umsetzung des sperrfreien Lesens

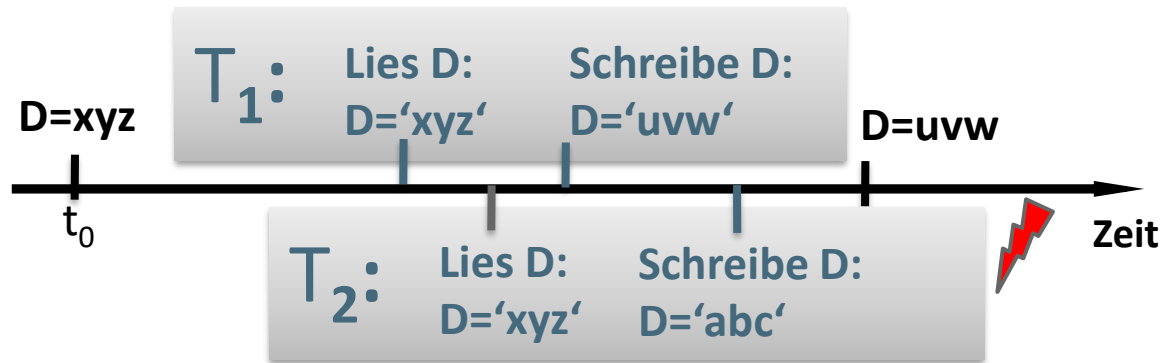


- T_2 erzeugt ausgehend von $v0$ ('xyz') neue Version $v1$ ('uvw').
Nach Commit von T_2 ist $v1$ die neue aktuelle Version des Datensatzes.
- Operationen in T_1 arbeiten mit der zum aktuellen Zeitpunkt gültigen Version, also anfangs mit $v0$ und nach Commit von T_2 mit $v1$ (engl. **Statement-level Snapshot**).
- Statement-level Snapshot entspricht READ COMMITTED, während Transaction-Level Snapshot schließt auch „non-repeatable reads“ aus (REPEATABLE READ)

MVCC

Schreibkonflikt

- Konflikte treten nur auf, wenn Transaktionen dieselbe Ausgangsversion überschreiben wollen:



- Beim Festschreiben wird die beim Start gültige Ausgangsversion mit der aktuellen Version verglichen. Falls die gelesene Ausgangsversion älter als die aktuelle ist, wird die Transaktion zurückgerollt und wiederholt.
- T_2 wird dementsprechend zurückgesetzt (Rollback) und kann ausgehend von der neuen aktuellen Version ('uvw') wiederholt werden.

Es gibt verschiedene Ansätze zur notwendigen **Versionierung** einzelner Datensätze:

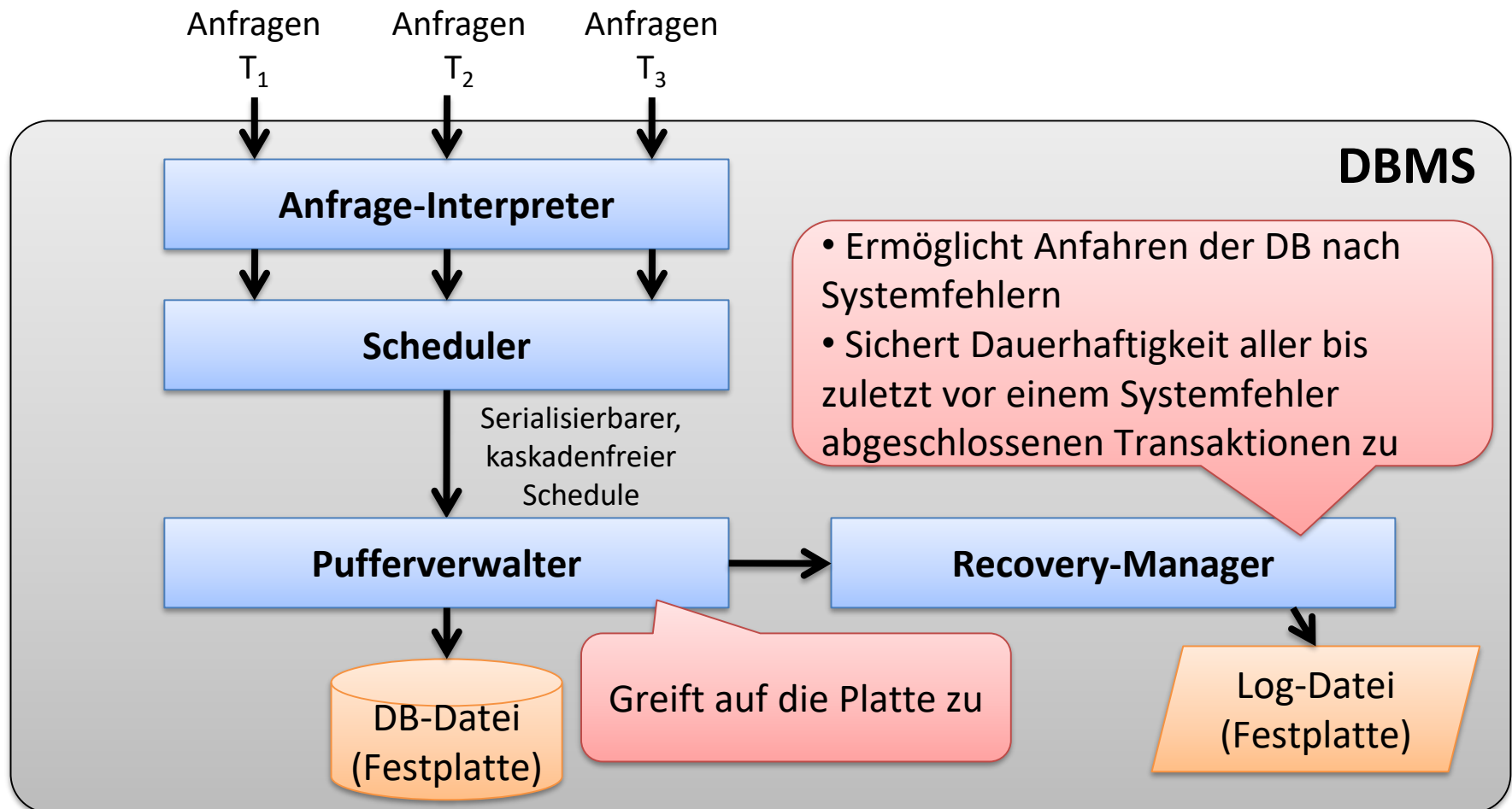
- Mittels monoton steigender **Transaktionsnummern**:
Ältere Transaktion erzeugt ältere Versionen
- **Startzeitpunkt** (Zeitstempel) der schreibenden Transaktion dient als Versionsnummer, oder
- **Zeitpunkt des Schreibvorgangs**, welcher die neue Version erzeugte
- Neben der Versionierungsinformation enthalten Datensätze auch einen Verweis auf die jeweilige **Vorgängerversion**, um Schreib-Konflikte erkennen und ggfs. auflösen zu können.
- In gewissen Zeitabständen müssen alte, d.h. von keiner Transaktion mehr benutzte, Versionen aus dem System **entfernt** werden.
- Manche DBMS erlauben den Zugriff auf unterschiedliche Versionen desselben Datensatzes und ermöglichen somit sog. „**Zeitreisen**“.

3. Fehlerbehandlung von Transaktionen

Sicherstellung der Dauerhaftigkeit von Transaktionen in DBMS

Protokollierung und Wiederanlauf

- **Recovery Manager:** Protokolliert relevante Ereignisse in laufenden Transaktionen und steuert nach Systemfehlern die Wiederherstellung eines korrekten Datenbankzustandes.



Sicherstellung der Dauerhaftigkeit von Transaktionen in DBMS

Recovery-Manager

- Der **Recovery-Manager** stellt gemäß **WAL-Prinzip** (*Write-Ahead Logging*) die **reihenfolgetreue Persistierung** aller **schreibenden Operationen** in einer oder mehreren **Log-Dateien** sicher.
- **WAL-Prinzip:** Alle Änderungen werden in den Log-Dateien persistiert, **bevor** diese in der Datenbank (DB-Datei) persistiert werden.
- **Log-Dateien** werden parallel zur DB-Datei als zusätzlicher **dauerhafter Speicher**, i.d.R. auf anderen Festplatten, betrieben.

Je nach Strategie der Pufferverwaltung speichert eine Log-Datei:

- **Before-Image für Undo:** Zustände von Datensätzen, bevor sie von noch aktiven oder abgebrochenen TA verändert wurden (s. **Steal**)
- **After-Image für Redo:** Änderungen abgeschlossener Transaktionen, sofern sie noch nicht persistiert wurden (s. **No-Force**).

Sicherstellung der Dauerhaftigkeit von Transaktionen in DBMS

Protokolleinträge

Typische Struktur von Logeinträgen bei *No-Force* und *Steal*:

[LSN, TRID, PageID, Redo, Undo, PrevLSN]

- **Log Sequence Number (LSN):** Eindeutige Kennung des Logeintrags
 - **LSN wird monoton aufsteigend vergeben**, um die zeitliche Reihenfolge der Protokolleinträge eindeutig festzuhalten
- **Transaktions-ID (TRID):** Kennung der TA, die die Operation durchführte
- **PageID:** Kennung der Seite, auf welche die Änderungsoperation abzielte
 - **Achtung:** Wenn eine Operation mehr als eine Seite betrifft, muss pro Seite ein Log-Eintrag generiert werden!
 - Seiten bzw. Blöcke enthalten ihrerseits die **jüngste** sie betreffende LSN.
- **Redo (After-Image):** Wie kann die Änderung nachvollzogen werden
- **Undo (Before-Image):** Wie kann die Änderung rückgängig gemacht werden
- **PrevLSN:** Zeiger auf den vorhergehenden Logeintrag der **gleichen** TA und Seite. Ziel: Schnelle Rekonstruktion der Schritte beim evtl. Recovery.

Sicherstellung der Dauerhaftigkeit von Transaktionen in DBMS

Rekonstruktion mittels Before- und After-Images

Zwei Strategien der Protokollierung, die i.d.R. kombiniert werden:

- **Physische Protokollierung von Werten** der geänderten Datensätze:
 - **Before-Image** hält Zustand des Datums vor der TA-Operation fest
 - **After-Image** speichert Zustand des Datums nach der TA-Operation
 - Welche Vor- und Nachteile sehen Sie?

- **Logische Protokollierung von Operationen** an Datensätzen:
 - **Before-Image** speichert **Undo-Code** um den vorherigen, gültigen Zustand des betroffenen Datensatzes zu **berechnen**.
 - **After-Image** enthält **Redo-Code** um den neuen gültigen Zustand des Datensatzes zu berechnen.
 - Welche Vor- und Nachteile sind erkennbar?

Beispiel für logische Protokollierung

Logeinträge einer Transaktion

Pseudo-Log zu unserer Beispieltransaktion *TRX-4711* zur Umbuchung:

- Belaste Girokonto A mit 50 Euro, erhöhe Sparkonto B um 50 Euro
- Beachte abweichende Struktur bei BOT (#1.) und COMMIT (#8.)

#	Operationen	Log-Einträge
		[LSN, TRID, PageID, Redo , Undo , PrevLSN]
1.	BOT	[#001, TRX-4711, BOT]
2.	r(A, a)	-
3.	a := a-50	-
4.	w(A, a)	[#002, TRX-4711, P _A , A-=50 , A+=50 , #001]
5.	r(B, b)	-
6.	b := b+50	-
7.	w(B, b)	[#003, TRX-4711, P _B , B+=50 , B-=50 , #002]
8.	COMMIT	[#004, TRX-4711, Commit, #003]
...

Sicherstellung der Dauerhaftigkeit von Transaktionen in DBMS

Verknüpfung der Daten mit Protokolleinträgen

Um im Fehlerfall den korrekten Zustand einer Seite **wiederherstellen** (physische Prot.) oder **berechnen** (log. Prot.) zu können, müssen Seiten/Blöcke und Logeinträge miteinander *verknüpft* werden:

- Jede **Seite** enthält die sie betreffende **jüngste LSN** in einem hierfür reservierten Feld der Satztabelle (s. Teil I, RDBMS).
- Beim Verdrängen der Seite aus dem Hauptspeicher (DB-Puffer) wird diese Information zusammen mit allen übrigen Daten in den jeweiligen Festplattenblock der DB-Datei kopiert.
- Somit „weiß“ das DBMS, ob in der persistierten Datenbank das Before- oder das After-Image einer Seite gespeichert ist:
 - Falls LSN des Blocks auf Festplatte < aktuelle LSN des Blocks in der Logdatei, speichert die Datenbank das Before-Image
 - Falls LSN im persistenten Block = aktuelle LSN in Log, enthält die Datenbank das After-Image.

Sicherstellung der Dauerhaftigkeit von Transaktionen in DBMS

Was passiert im Fehlerfall?

Fehlertyp 1: Eine Transaktion scheitert z.B. wegen Deadlock-Auflösung

- Sämtliche Änderungen dieser TA werden mittels Before-Images in der nach *LSN absteigenden* Reihenfolge zurückgesetzt/berechnet (Undo).
- Mittels *PrevLSN* sind die relevanten Logeinträge schnell gefunden
- Wenn während der Transaktion **Savepoints** gesetzt wurden, müssen deren Auswirkungen ebenfalls rückgängig gemacht werden, d.h. Undo läuft bis BOT-Eintrag gefunden ist.
- Falls andere Transaktionen die betroffenen und nun zurückgesetzten Seiten zwischenzeitlich gelesen und überschrieben haben, müssen jene Transaktionen ebenfalls zurückgesetzt werden („Kaskade“).

Sicherstellung der Dauerhaftigkeit von Transaktionen in DBMS

Was passiert im Fehlerfall?

Fehlertyp 2: Hauptspeicherverlust (z.B. OS-, RAM-Fehler, ...)

1. Log-Analyse: Beim Anfahren werden anhand der Log-Datei (Commit-Eintrag vorhanden?) abgeschlossene und nicht abgeschlossene TA erkannt

2. Redo-Phase für **alle (!)** Transaktionen:

- Für jeden Logeintrag -nach LSN aufsteigend- wird LSN im Logeintrag mit der LSN des zugehörigen Blocks auf DB-Festplatte verglichen
- Wenn $LSN-in-Log > LSN-in-Block$ wird **Redo** ausgehend vom After-Image durchgeführt und $LSN-in-Block = LSN-in-Log$ gesetzt.

3. Undo-Phase nur für **nicht erfolgreiche** Transaktionen:

- Logeinträge nach LSN absteigend durchsuchen und für alle TA ohne Commit-Eintrag **Undo** Operationen durchführen
- Dies geschieht unabhängig von LSNs der persistierten Blöcke:
 - Daten sind entweder durch „Steal“, oder in obiger Redo-Phase entstanden
 - Undo-Operationen müssen ebenfalls im Log protokolliert werden (sog. Compensation Log Records)