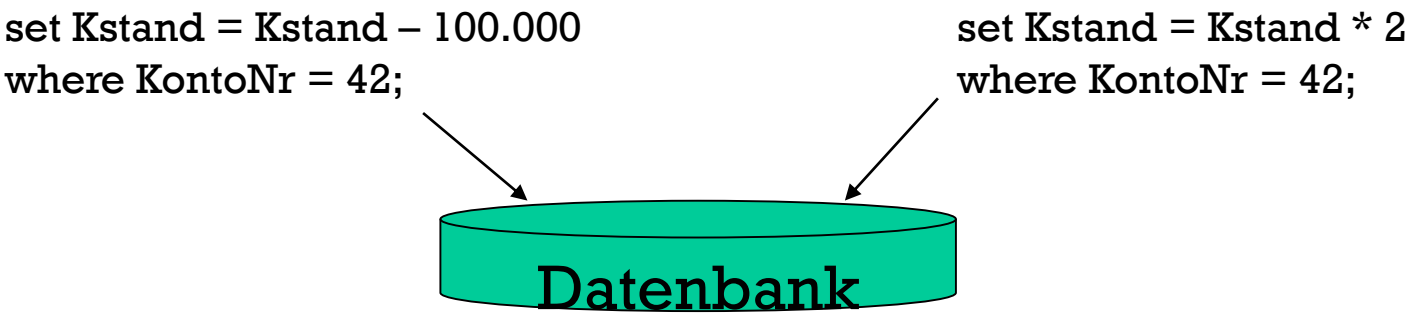
5. Transaktionen

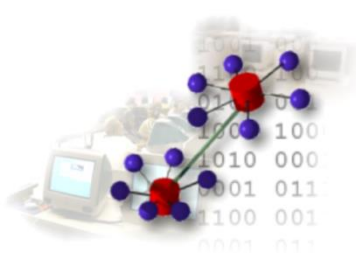
- **Motivation**
- **Definition**
  - ACID-Bedingungen
  - Synchronisationsprobleme
- **Serialisierung**
- **Synchronisationsverfahren**
  - 2PL
- **Snapshot-Isolation**
- **Fehlerbehandlung**

Motivation: Mehrbenutzerbetrieb

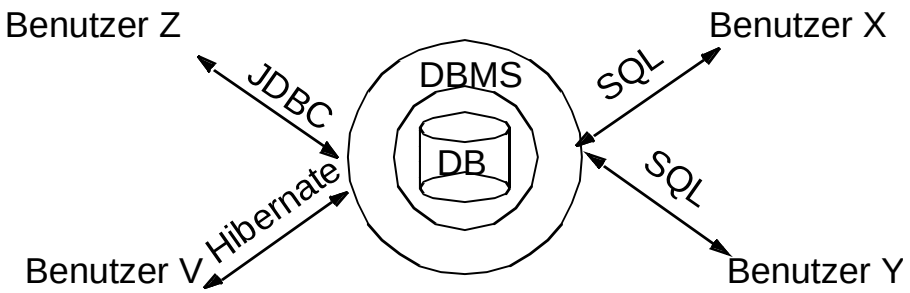
- **Datenbanken sind ein integrierter Datenbestand mehrerer Benutzer.**

- **Problem**
  - Wie funktioniert der gleichzeitige Zugriff auf die Datenbank durch mehrere Benutzer?





Mehrbenutzerbetrieb



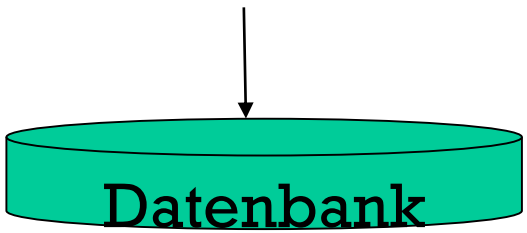
- Mehrbenutzerbetrieb:
  - DBS bedient mehrere Benutzer gleichzeitig.
  - Im Extremfall: verschiedene Benutzer greifen auf gleichen Datensatz zu.
- Aktivität eines Benutzers:
  - sequentieller Prozess
- Aktivitäten mehrerer Benutzer:
  - Menge ineinander verzahnt ablaufender Prozesse auf einer gemeinsamen Datenbasis

Motivation: Atomare Ausführung

- **Bündelung mehrerer Operationen zu atomaren Ausführungseinheiten.**
  - Kontoüberweisung erfordert zwei Änderungsoperationen

set Kstand = Kstand – 100.000 where KontoNr = 42;

set Kstand = Kstand + 100.000 where KontoNr = 77;



Motivation: Umgang mit Fehlern

- **Umgang mit Fehler**
  - Was passiert z. B. im Fall eines Systemabsturzes, bei dem der Hauptspeicher verloren geht?
  - Ist meine Datenbank noch in Ordnung?

Datenbank nach Fehler

- Befindet sich die Datenbank auch nach einem Absturz in einem konsistenten Zustand?
  - Alle Konsistenzbedingungen sollten erfüllt sein.
- Sind alle Änderungen auch wirklich in der Datenbank?
  - Eine erfolgreiche Buchung ist auch tatsächlich in der Datenbank wirksam geworden.

5.1 Transaktion

- Die Probleme können durch Transaktionen behoben werden.

- Eine Transaktion besteht aus Elementaroperationen:
  - Operationen mit Bezug zur Datenbank
    - Lesen und Schreiben
  - Elementaroperationen ohne Auswirkung auf die Datenbank.
    - Operationen in einem AWP ohne Bezug zur Datenbank
  - Operationen für den Kontrollfluss einer Transaktion.

Operationen in Transaktionen

- Elementaroperationen mit Bezug zur Datenbank
  - Lesen des Werts eines Objekts A aus der Datenbank in eine Programmvariable a:
    read(A,a) bzw. **r(A)**
  - Zuweisung eines Werts einer Programmvariable a an ein Objekt A der Datenbank
    write(A, a) bzw. **w(A)**
- Ablaufsteuerung
  - Anfang einer Transaktion: **bot**
    - Oft nicht explizit benutzt, da nach dem Beenden einer Transaktion die nächste Transaktion implizit beginnt.
  - Erfolgreiches Ende der Transaktion: **commit**
    - Alle in der Transaktion erzeugten Änderungen werden in der Datenbasis festgeschrieben.
  - Abbruch einer Transaktion: **abort**
    - Alle in der Transaktion vorgenommen Änderungen der Datenbasis werden unwirksam.

Aufbau einer Transaktion

- **Eine Transaktion (TA) T ist eine Folge von Elementaroperationen.**
  - Eine Transaktion startet mit T.bot().
  - Eine Transkation wird beendet mit T.c() oder T.a().
    - c() steht für commit
    - a() steht für abort
  - Ansonsten bezeichnen wir mit T.read(A,a) bzw. T.r(A) und T.write(A,a) bzw. T.w(A) die Elementaroperationen mit Auswirkung auf das Objekt A der Datenbank.
  - Eine Transaktion erfüllt die **ACID-Bedingungen**.

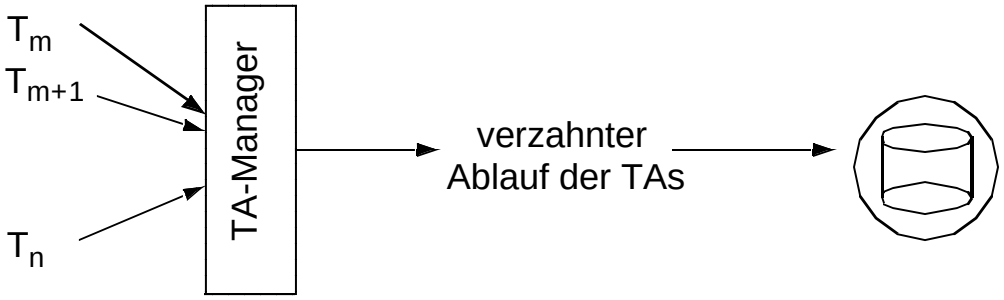
ACID-Bedingungen

- **A**: TA ist die kleinste, **atomare Ausführungseinheit.**
  - Entweder werden alle durch einen TA vorgenommenen Änderungen in der Datenbasis wirksam oder gar keine.
- **C**: Eine TA überführt einen konsistenten Datenbankzustand in einen anderen **konsistenten Datenbankzustand.**
  - Innerhalb einer TA sind Inkonsistenzen erlaubt.
- **I**: Eine TA ist gegenüber anderen TAs **isoliert, d. h. das Ergebnis einer TA kann nicht direkt durch eine andere TA beeinflusst werden.**
  - Jede TA wird logisch so ausgeführt, als gäbe es keine andere TA.
- **D**: Ist eine TA erfolgreich abgeschlossen, dann bleibt ihre Wirkung auf die Datenbasis **dauerhaft erhalten.**
  - Dies gilt auch im Fall eines Systemfehlers (Verlust des Hauptspeichers).

Transaktionen in der SQL-Shell von PostgreSQL

- **Starten einer Transaktion**
  - BEGIN TRANSACTION
- **Erfolgreicher Abschluss einer Transaktion**
  - COMMIT
- **Zurücksetzen einer Transaktion**
  - ROLLBACK
- **Lesen eines Objekts**
  - select * from personal where pnr = 67;
- **Ändern eines Objekts in der Datenbank**
  - Z. B. Löschen;
    - delete * from pmzuteilung where pnr = 67

Transaktionsmanagement



- **Aufgaben des Transaktionsmanagers**
  - Synchronisation der TAs
    - Isolation

    ➔ Einschränkung der verzahnten Abläufen
  - Zurücksetzen einer oder mehrerer TAs
    - Atomarität
    - Dauerhaftigkeit
    - Konsistenz

    ➔ Einschränkung der verzahnten Abläufen

Historie: Reihenfolge von Operationen

- Für einen Ablauf einer einzelnen TA T gibt es eine Ordnungsrelation $<_T$, welche die sequentielle Ordnung der Elementaroperationen ausdrückt:
  $op_1 <_T op_2 \quad \Leftrightarrow \quad op_1$ wird vor op_2 ausgeführt.

- **Definition (Ausführungsplan, Historie)**
  - Seien $T_1,\ldots,T_n$ Transaktionen. Dann wird eine Folge H **aller** Operationen der TAs $T_1,\ldots,T_n$ ein Ausführungsplan genannt, falls folgende Bedingungen erfüllt sind:
    - Es gibt nur Elementaroperationen vom Typ r, w, a und c.
    - Die Ordnungsrelationen $<_{Tj}$ aller Transaktionen bleiben bewahrt.

- **Bemerkungen**
  - Der Ausführungsplan definiert eine Ordnungsrelation $<_H$.
  - Nicht alle Ausführungspläne erzeugen einen konsistenten Datenbankzustand (siehe Beispiele)

Beispiel

Transaktion T_1:

read(A, a);

a := a + 10;

write(A, a);

read(B, b);

b := b-10;

write(B, b);

commit;

Transaktion T_2:

read(B, b);

b := b + 10;

write(B, b);

read(C, c);

c := c-10;

write(C, c);

commit

Paralleler Ablauf von T1 und T2

T1.read(A, a);

T2.read(B, b);

a:= a + 10;

T1.write(A, a);

b := b + 10;

T1.read(B, b);

T2.write(B, b);

b := b-10;

T2.read(C, c);

T1.write(B, b);

T1.commit();

c := c-10;

T2.write(C, c);

T2.commit();

Zeit

Ausführungsplan:

T1.r(A);

T2.r(B);

T1.w(A);

T1.r(B);

T2.w(B);

T2.r(C);

T1.w(B);

T1.c();

T2.w(C);

T2.c();

Synchronisationsprobleme

- Ein Ausführungsplan muss gewisse Kriterien erfüllen, damit die Isolationseigenschaft garantiert ist. Sonst kann es folgende Probleme geben:
  - Lost update
  - Inkonsistente Sicht auf die Datenbank
  - Inkonsistente Datenbank
  - Phantome

Problem “lost update”

- Transaktion T_1 und T_2 erhöhen das Gehalt eines Mitarbeiters jeweils um 100,- bzw. 200,+ Euro.
- Ausführungsplan

```
T1.r(Gehalt);
T2.r(Gehalt);
// T1 erhöht Gehalt um 100
T1.w(Gehalt);
T1.c ();
// T2 erhöht Gehalt um 200
T2.w(Gehalt);
T2.c ();
```

- Bemerkungen
  - Konsistenz der DB ist i.a. nicht verletzt
  - Resultate der Anfragen sind nicht “offenkundig falsch”

Problem der inkonsistenten Sicht auf die Datenbank

- A und B sind Kontostände für die A+B=0 gilt.
  - T_1 und T_2 sind zwei Transaktionen, wobei T_1 ändert den Kontostand von A und B und T_2 liest die Kontostände.
- Ausführungsplan

  T1.r(A);

  // T1 dekrementiert den Kontostand von A um 1

  T1. w(A); // Zurückschreiben des geänderten Werts

  T2. r(A);

  T2. r(B); // T2 erkennt, dass A + B ≠ 0 gilt.

  T2.c();

  T1.r(B);

  // T1 erhöht den Kontostand von B um 1

  T1. w(B);

  T1. c();
- Problem
  - Transaktion T_2 sieht inkonsistente Daten Datenbank, obwohl die Datenbank irgendwann wieder konsistenten Zustand ist.

Problem der inkonsistenten Datenbank

- A und B sind zwei Kontostände, die A = B erfüllen.
  - T_1 erhöht die beiden Kontostände um 10 Euro
  - T_2 erhöht die beiden Kontostände um 10%
- Ausführungsplan

```
T1. r(A);   // A wird um 10 erhöht
T1. w(A);
T2. r(A);   // A wird um 10% erhöht
T2. w(A);
T2. r(B);   // B wird um 10% erhöht
T2.w(B);
T2.c();
T1.r(B);                  // B wird um 10 erhöht
T1. w(B);
T1. c();
```

Problem

- Datenbank ist dauerhaft in einem inkonsistenten Zustand:
  $A_{neu} = (A+10)*1.1 \neq B*1.1+10 = B_{neu}$

Phantom-Problem

- Für die Angestellten eines Unternehmens soll berechnet werden, wie hoch die Gewinnausschüttung ist.
  - Jeder Angestellte soll ein Anteil bekommen.
- Transaktion T_1
  1. Berechnung des Gewinnanteils pro Angestellter
     - liest die Daten aller Angestellten,
     - berechnet die Höhe des Gewinnanteils
  2. Zuweisung der Bonuszahlungen an alle Angestellten
- Transaktion T_2
  - fügt einen neuen Angestellten ein („Phantom“).

- **Problem**
  - Wird T_2 nach Schritt 1 von T_1 ausgeführt, so ist die Kalkulation von T_1 veraltet.

5.2 Serialisierung von TAs

Sei T* eine Menge von Transaktionen und H ein dazugehörender Ausführungsplan. Seien T1, T2 $\in$ T*, zwei Transaktionen, die gemeinsam auf ein Datenobjekt A zugreifen. Es werden folgende vier Fälle unterschieden:

1. $T1.r(A) <_H T2.r(A)$
2. $T1.r(A) <_H T2.w(A)$
3. $T1.w(A) <_H T2.r(A)$
4. $T1.w(A) <_H T2.w(A)$

- Nur im 1. Fall sind die Operationen vertauschbar, ohne dass sich das Ergebnis des Ausführungsplans ändert.
- In allen anderen Fällen ist davon auszugehen, dass ein Vertauschen der Ausführungsreihenfolge zu einem anderen Ergebnis führt. Man spricht dann auch von einem **Konflikt zwischen T1 und T2**.

Äquivalenz von Ausführungsplänen

- **Definition:**
  - Sei T* eine Menge von Transaktionen. Seien H und G zwei dazugehörige Ausführungspläne für T*. Dann sind H und G **äquivalent,** falls ihre Konflikte identisch sind. Für alle Transkationen T1, T2 $\in$ T* und ein beliebiges Datenobjekt A gelten folgende Bedingungen:
    - **$T1.r(A) <_H T2.w(A) \Leftrightarrow T1.r(A) <_G T2.w(A)$**
    - **$T1.w(A) <_H T2.r(A) \Leftrightarrow T1.w(A) <_G T2.r(A)$**
    - **$T1.w(A) <_H T2.w(A) \Leftrightarrow T1.w(A) <_G T2.w(A)$**

- **Beobachtung**
  - Durch Vertauschen von zwei benachbarten Operationen, die nicht in Konflikt zueinander stehen, kann man sich einen äquivalenten Ausführungsplan erzeugen.

Beispiel

- Transaktionen T1 und T2
- Ausführungsplan H =
  (T1.r(A),T2.r(C),T1.w(A),T2.w(C),T1.r(B),T1.w(B),T1.c(),
  T2.r(A),T2.w(A),T2.c())
- Konflikte:
  $T1.r(A) <_H T2.w(A)$
  $T1.w(A) <_H T2.r(A)$

Serialisierbare Ausführungspläne

- Bei einem **sequentiellen** Ausführungsplan H laufen die Operationen einer Transaktion komplett vor oder nach den Operationen einer anderen Transaktionen ab.
- Beispiele
  - (T1.r(A), T1.w(A), T1.r(B),T1.w(B),T1.c(), T2.r(C), T2.w(C), T2.r(A),T2.w(A),T2.c())
  - (T2.r(C), T2.w(C), T2.r(A),T2.w(A),T2.c(), T1.r(A), T1.w(A), T1.r(B),T1.w(B),T1.c())

Definition (Serialisierbarkeit):

- Ein Ausführungsplan ist **serialisierbar**, falls es einen äquivalenten sequentiellen Ausführungsplan gibt.

Serialisierbarkeitsgraph

- Problem
  - Wie kann ich für eine Menge T* von Transkationen und eine Historie H effizient die Serialisierbarkeit testen?
- **Serialsierbarkeitsgraphen** G_H
  - Zu einer Historie H definieren wir den **gerichteten** Graph G_H = (K,U) mit Knotenmenge K und Kantenmenge $U \subseteq K \times K$
    - zu jeder TA $T \in T^*$ gibt es genau einen Knoten mit dem Label T
    - $(T1,T2) \in U \Leftrightarrow$ es gibt in H einen Konflikt zwischen T1 und T2.

Theorem

- Ein Ausführungsplan H ist genau dann serialisierbar, falls G_H zyklenfrei ist.

Beweis:

- siehe Bernstein, Hadzilacos, Goodman.: Concurrency Control and Recovery in Database Systems. Addison-Wesley 1987.

5.3 Synchronisationsverfahren

- **Höchste Anforderung an ein DBS**
  - Garantie der Serialisierbarkeit

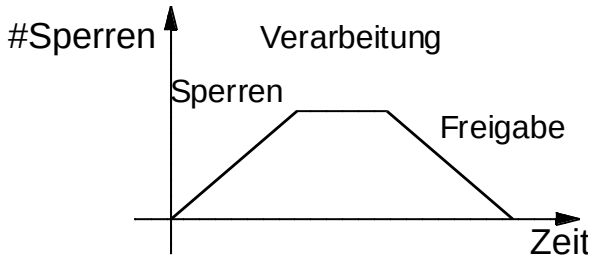
- **Unterscheidung zwischen folgenden Verfahren**
  - präventive Verfahren:
    - Verhindere stets nicht-serialisierbare Ausführungspläne.
    - Bislang verwendete Verfahren in der Praxis
      - Sperrverfahren
      - Zeitstempel- und Mehrversionsverfahren
  - verifizierende (“optimistische”) Verfahren:
    - Beobachte ständig die Ausführungspläne (über den Graph G).
    - Falls Serialisierbarkeit verletzt wird, setze eine TA zurück und starte sie neu.

Sperrverfahren

- Jede TA sperrt den Teil der DB, auf dem sie arbeitet.
  - Solange (exklusiv) gesperrt ist, können keine anderen TAs zugreifen.
- Klassifizierung der Sperrverfahren nach
  - Sperrobjekten: Datensatz, Datenseite, Relation, Datenbank
    - je feiner die Sperreinheit, desto mehr Parallelität
    - je feiner die Sperreinheit, desto mehr Verwaltungsaufwand
  - Sperrmodi
    - lock(A) sperrt das Datenobjekt A
    - unlock(A) gibt die Sperre auf A wieder frei
  - Sperrprotokoll

5.3.1 Zwei-Phasen Sperrprotokoll (2PL)

- **Regel**
  - Bevor ein Objekt in einer TA genutzt wird, muss zuvor ein Lock auf dem Objekt gesetzt werden.
  - Für jede TA darf nach dem ersten unlock kein lock mehr angefordert werden.



- **Eigenschaften**
  - Das gleiche Objekt darf während einer Transaktion bei einem 2PL nur einmal gesperrt und freigegeben (unlock) werden.
  - Wurde das Objekt verändert, muss es vor der Freigabe (unlock) auch „geschrieben“ werden.

2PL-Historien sind serialisierbar

Theorem

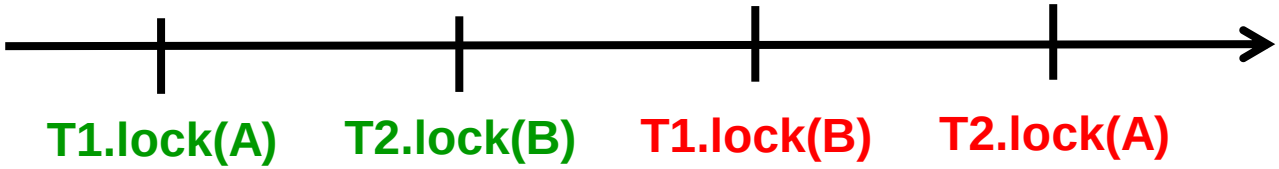
- Jeder durch ein 2-Phasen Sperrprotokoll erzeugte Ausführungsplan ist serialisierbar.

- **Beweisskizze:**
  - Widerspruchsannahme: H ist ein Ausführungsplan mit einem Zyklus $T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$
  - Wenn dieser mittels eines 2PL entstanden wäre, dann müsste es Objekte $A_1, \ldots, A_n$ geben, so dass Transkationen $T_1,\ldots,T_n, T_1$ bzgl. diesen Objekten in Konflikt stehen.
  - Somit muss also ein T_j.unlock(A_j) vor einem T_{j+1}.lock(A_j) für j= 1, …,n-1 erfolgt sein und zusätzlich noch ein T1.lock (A_n) nach dem T_n.unlock(A_n) erfolgen.

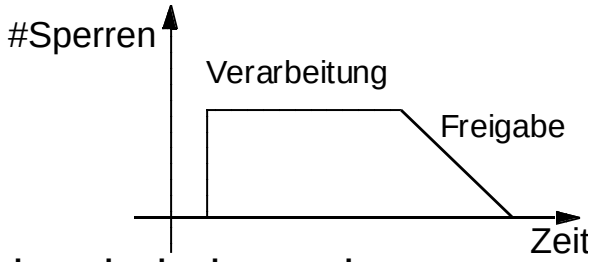
    ➔ In T1 wird ein Objekt nach einem unlock gesperrt.

Variante 1: konservatives 2PL

- Problem
  - Verklemmungen sind möglich



- Lösung
  - Alle möglichen Sperren einer Transaktion T werden am Anfang von T gesetzt.

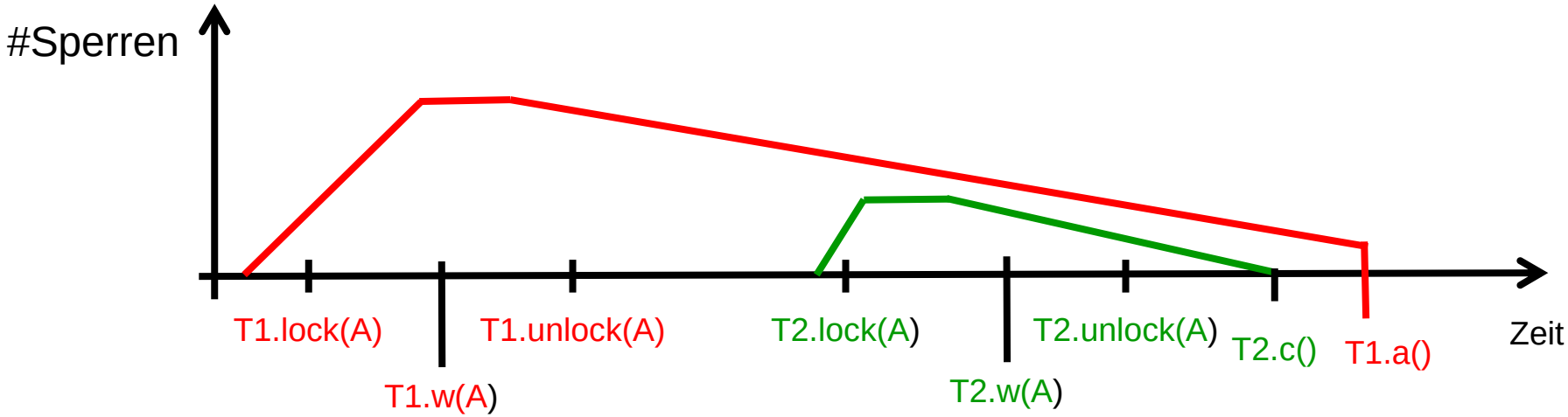


  - Zwar ist das Problem behoben, aber
    - Objekte müssen zu Beginn von T bereits bekannt sein.
    - Häufig wird zu viel und zu lange gesperrt.

➔ Stattdessen wird eine pragmatische Lösung bevorzugt.
Welche?

Kaskadierendes Zurücksetzen

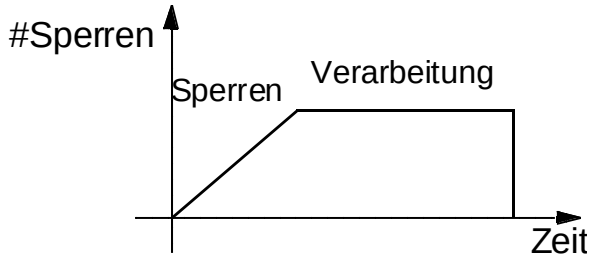
- **Problem**
  - Kaskadierendes Zurücksetzen



  - Widerspruch zu der Forderung nach Dauerhaftigkeit bei Transaktionen.

Striktes 2PL

- Lösung zur Vermeidung von kaskadierendem Zurücksetzen
  - Alle Sperren einer Transaktion T werden bis zum Schluss von T gehalten und erst dann zusammen freigegeben.



- Dieses Protokoll wird in nahezu allen relationalen Datenbanksystemen unterstützt, die Locking verwenden.
  - Optional können beim Lesen die Locks bereits vor dem Ende der Transaktion freigegeben werden.

Sperrmodi

- **Problem**
  - Keine Unterscheidung beim Sperren, ob das Objekt geschrieben oder nur gelesen wird.
  - ➔ Zu restriktive Vorgehensweise
  - ➔ Einschränkung der Parallelität
- **Lösung: RX-Protokoll**
  - Unterscheidung zwischen zwei Sperrmodi
    - S-Sperre: T.slock(A)
      - Das Objekt A darf von T nur gelesen, aber nicht geschrieben werden.
      - Mehrere S-Sperren pro Objekt möglich.
    - X-Sperre: T.xlock (A)
      - Transaktion T hat nach dem Sperren des Objekts A exklusiven lesenden und schreibenden Zugriff.
      - Es ist somit nur eine X-Sperre auf einem Objekt A erlaubt .

5.3.2 Mehrbenutzersynchronisation in SQL-Datenbanken

- **In SQL kann das Korrektheitskriterium für die parallele Verarbeitung von Transaktionen durch Angabe des Isolationslevel abgesenkt werden.**
  - Vorteil: Erhöhung des Parallelitätsgrads
  - Nachteil: Gefahr einer inkonsistenten Datenbank
- **Diese Isolationslevel sind zwar im Standard verankert, aber**
  - Spezifikation ist teilweise zu ungenau
  - nicht alle Datenbanksystemhersteller unterstützen die verschiedenen Isolationslevels
    - z. B. Oracle und PostgreSQL.
  - Teilweise gibt es noch weitere Isolationslevel
    - z. B. in SQL-Server

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., & O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record* (Vol. 24). ACM.

Isolationslevel im SQL-Standard

- **Es gibt vier Isolationslevel im Standard**
  - **read uncommitted**
  - **read committed**
  - **repeatable reads**
  - **serializable**
    - Dies entspricht dem in diesem Kapitel erläuterten Serializierbarkeitsbegriff.
    - Dies ist i. A. die Defaulteinstellung bei DBS.
- **Jedoch haben sich diese Level nicht wirklich als praktikabel erwiesen.**
    - Insbesondere wenn man Datenbanksysteme nutzt, die nicht ein 2-Phasen Sperrprotokoll verwenden.
    - Man muss also bei jedem System genau schauen, was aus dem Standard umgesetzt wurde und wie.

Isolationslevel – read uncommitted

- **Leseoperationen in Transaktionen ignorieren jegliche Sperren von anderen Transaktionen.**
- **Konsequenzen**
  - Das Problem des lost-update kann auftreten, wenn die Transaktion nicht nur die Daten liest, sondern auch wirklich wieder zurückschreibt.
    - Diese Eigenschaft wird in den meisten Datenbanksystemen kritisch gesehen.
      - Dirty-reads werden z. B. dann nur bei lesenden Transaktionen erlaubt.
  - Dadurch kann eine Transaktion T_1 ein von einer Transaktion T_2 geändertes Datenobjekt lesen, die danach mit T_2.a() abgebrochen wird.
    - Der Zustand des Objekts wird wieder zurückgesetzt.

Isolationslevel – read committed

- **Transaktionen setzen Schreibsperren und halten diese Sperren bis zum commit.**
- **Beim Lesen werden Lesesperren nur kurz gehalten (bis das Objekt gelesen wurde).**

- **Konsequenzen**
  - Beim mehrmaligen Lesen des **gleichen** Objekts in einer Transaktion kann es zu unterschiedlichen Ergebnissen kommen.

    → non-repeatable reads

Isolationslevel - repeatable reads

- **Sowohl Lese- als auch Schreibsperren werden bis zum Ende der Transaktion gehalten.**

- **Konsequenzen**
  - Eine Transaktion sieht immer den gleichen Zustand eines Objekts in der Datenbank.
  - Während einer Transaktion neu eingefügte Objekte (einer anderen Transaktionen) werden aber bei Anfragen sichtbar.
    - Gleiche Anfragen in einer Transaktion können deshalb verschiedene Ergebnismengen liefern.

Kompakt als Tabelle

	Dirty Reads	Non-Repeatable Reads	Phantoms
Serialization	Nein	Nein	Nein
Repeatable Read	Nein	Nein	Möglich
Read Commited	Nein	Möglich	Möglich
Read Uncommited	Möglich	Möglich	Möglich

- **Dirty Writes**
  - Nicht möglich bei Repeatable Read und Serialization
  - Bei Read Committed und Read UnCommitted sollte unbedingt darauf geachtet werden, dass die Transaktionen nur lesend auf die Daten zugreifen.

5.3.3 Snapshot-Isolation

- **Ziele**
  - Erhöhung des Parallelitätsgrads bei Transkationen
  - Garantie der Serialisierbarkeit
    - Der Ablauf von ineinander verzahnt ablaufenden Transaktionen ist zu einem strikt seriellen Plan äquivalent.

- **Lösungen**
  - 2-Phasen-Sperrprotokoll
  - Zeitstempelverfahren
  - Multiversion-Verfahren
  - Snapshot-Verfahren
    - Derzeit in Oracle, …. und PostgreSQL verwendet.

Beispiel (SQL Server)

- **In SQL Server ist das strikte 2PL umgesetzt.**
  - Beispiel eines verzahnten Ablaufs (read uncommitted, 2 Transaktionen grün und rot)

```
begin transaction
(*)             set transaction isolation level read uncommitted
                begin transaction
insert into test values (8,8,8);
                select * from test where x > 3;  // Datensatz (8,8,8) ist sichtbar
                commit
rollback;
```

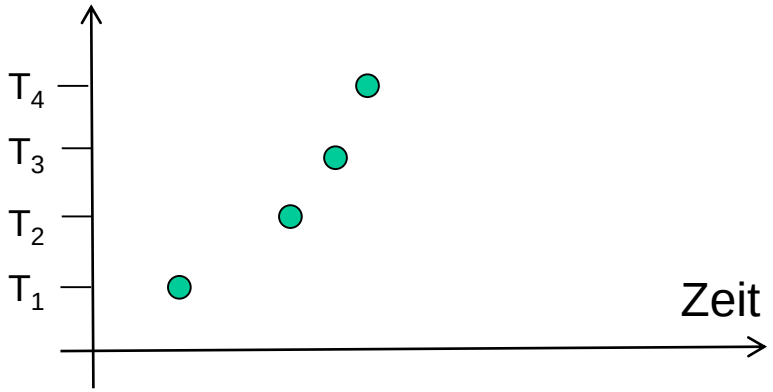
  - Was passiert, wenn in dem obigen Beispiel die Zeile (*) durch

```
set transaction isolation level read committed
```

    ersetzt wird?

5.3.3.1 Zeitstempel-Verfahren

- **Eine neu eintreffende Transaktion T_{neu} bekommt einen Zeitstempel ts zugewiesen, der größer als der Zeitstempel früherer Transaktionen T_i ist.**
  - Es gilt also $T_i.ts < T_{neu}.ts$ für alle im System laufenden Transaktionen T_i.

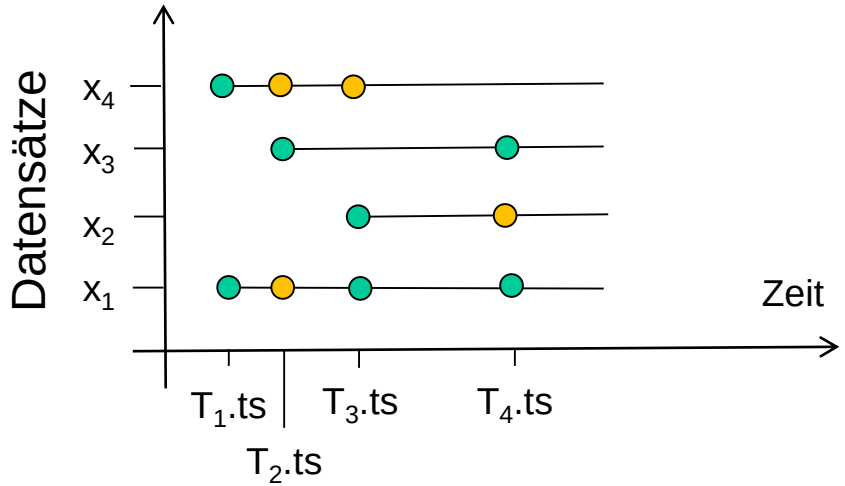


Ablaufplan: T_1, T_2, T_3, T_4

- **Der Ablaufplan der Transaktionen ist stets äquivalent zu der bzgl. den ts-Zeitstempeln definierten seriellen Ausführung.**

Zeitstempel in Datensätzen

- **Jeder Datensatz x besitzt zwei Zeitstempel:**
  - **x.write-ts** ist der Zeitstempel der Transaktion T, die x zuletzt erfolgreich geschrieben hat.
  - **x.read-ts** ist der größte Zeitstempel einer Transaktion, die X erfolgreich gelesen hat (Defaultwert = 0).



	Read-ts	Write-ts
x_1	T_2.ts	T_4.ts
x_2	T_4.ts	T_3.ts
x_3	0	T_4.ts
x_4	T_3.ts	T_1.ts

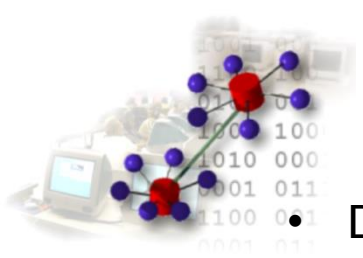
- **Die Algorithmen für das Lesen und Schreiben müssen die Ordnung der Transaktionen bewahren.**

Lesen von Datensätzen

- **Lesen eines Datensatzes x in einer Transaktion T**
  - Falls (T.ts < x.write-ts),
    - T soll ein Datensatz lesen, der bereits von einer anderen Transaktion mit größerem Zeitstempel überschrieben wurde.
    - Die Leseoperation wird abgelehnt und T mit abort abgebrochen.
  - Falls (T.ts ≥ x.write-ts),
    - **Lesen wird ausgeführt.**
    - x.read-ts := max(x.read-ts, T.ts).

Schreiben von Datensätzen

- **Schreiben eines Datensatzes x in einer Transaktion T**
  - Falls (T.ts < x.read-ts)
    - Dann hätte die Schreiboperation bereits vor der letzen Leseoperation erfolgen müssen.
    - Schreiboperation wird abgelehnt und die Transaktion mit abort abgebrochen.
  - Falls (T.ts < x.write-ts)
    - Schreiboperation hätte früher erfolgen müssen.
    - Schreiboperation wird abgelehnt und die Transaktion mit abort abgebrochen.
  - Ansonsten
    - Schreiboperation wird durchgeführt.
    - x.write-ts := T.ts



Beispiel

- Datensätze: v,x,y,z
- Transaktionen T_1, T_2, …, T_5

T_1	T_2	T_3	T_4	T_5
				T_5.r(x)
	T_2.r(y)			
T_1.r(y)				
		T_3.w(y)		
		T_3.w(z)		
	T_2.r(z)			
	??			
T_1.r(x)			T_4.r(v)	
			T_4.r(v)	
		T_3.w(v)		
		??		
				T_5.w(y)

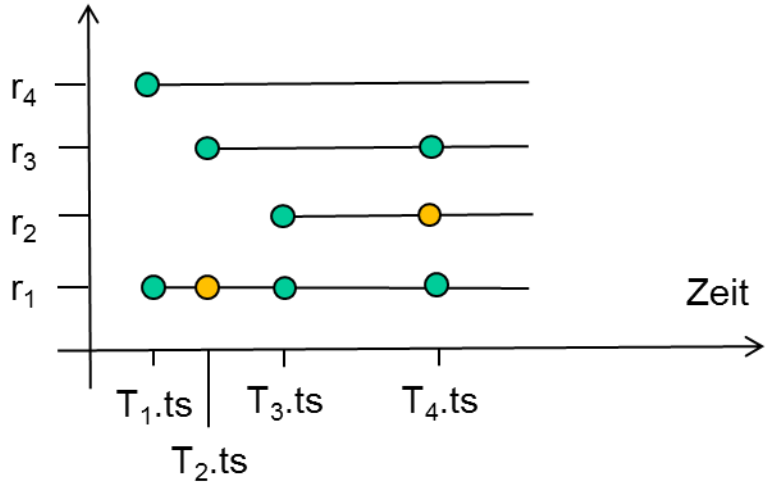
Zeit ↓

Eigenschaften des Zeitstempelverfahrens

- **Serialisierbarkeit ist erfüllt.**
  - Im Serialisierbarkeitsgraph ist der Zeitpunkt des Startknotens einer Kante stets kleiner als der Zeitpunkt des Endknotens.
- **Keine Deadlocks**
  - Transaktionen warten nicht auf die Freigabe von Ressourcen von anderen Transaktionen.
- **Kaskadierendes Zurücksetzen**
  - Dies ist möglich, da beim Abbruch einer Transaktion T auch alle anderen T‘ mit T‘.ts > T.ts zurückgesetzt werden müssen.

5.3.3.2 Multiversions-Verfahren

- **Multiversions-Verfahren (MV) erhalten die alten Versionen eines Datensatzes, um den Grad an Parallelität zu erhöhen.**
  - Multiversion Timestamp Ordering (MVTO)
  - Multiversion Two-Phase Locking (MV2PL)



Eigenschaften

- **Beim Lesen r(x) wird eine Version gewählt (in Abhängigkeit des Transaktionsstempels) und der entsprechende Wert geliefert.**
  - Leseoperation müssen nie warten, da stets sofort eine Version geliefert werden kann.
- **Jedes Schreiben führt zum Erstellen einer neuen Version.**
- **Zeitstempel werden als Label für die Versionen benutzt.**

MVTO

- **Jeder Datensatz *x* entspricht einer Folge von Versionen <x_1, x_2,...., x_m>.**
- **Jede Version Q_k enthält folgende Information**
  - **x_k.wert** – der Wert in der Version x_k.
  - **x_k.write-ts –** Zeitstempel der Transaktion, die x_k erzeugt hat.
  - **x_k.read-ts –** größter Zeitstempel einer Transaktion, die x_k erfolgreich gelesen hat.
- **Falls Transaktion *T eine neue Version x_k von x erzeugt.***
  - x_k.write-ts := x_k.read-ts := T.ts
- **Beim Lesen einer Version x_k durch Transaktion T**
  - x_k.read-ts : = max(T.ts, x_k.read-ts)

Lesen und Schreiben

- **Lesen von Datensatz x durch Transaktion T**
  - Wähle die Version k mit dem größten x_k.write-ts, so dass x_k.write-ts < T.ts noch erfüllt ist.
  - Es wird x_k zurückgegeben.

- **Schreiben von x durch Transaktion T**
  - x_k hat den größten x_k.write-ts, so dass x_k.write-ts < T.ts gilt.
    - Falls (T.ts < x_k.read-ts) T.a().
    - Falls (T.ts == x_k.write-ts) überschreibe den Inhalt von x_k.
    - Ansonsten, erzeuge eine neue Version.

- **Diese Algorithmen garantieren die Serialisierbarkeit von MVTO!**

MV2PL

- **Unterscheidung zwischen nur lesenden und schreibenden Transaktionen**
- ***Schreibende Transaktionen***
  - *Erwerben von Locks, die bis zum Ende der Transaktion gehalten werden.*
  - Jedes Schreiben erzeugt eine neue Version.
  - Jede Version besitzt nur einen Zeitstempel, der durch Zuweisung eines Zählers **ts-counter** generiert wird.
    - **ts-counter wird beim commit inkrementiert.**
- ***Lesende Transaktionen***
  - *Lesen den aktuellen Wert von ts-counter vor der Ausführung .*
  - Aufruf des Algorithmus für das Lesen beim MV-Zeitstempel-Verfahren.

5.3.4 Snapshot Isolation (SI)

- **Motivation**
  - In vielen Anwendungen gibt es viele lesende Transaktionen, die auf viele Daten zugreifen.
  - Diese stehen aber im Konflikt zu schreibenden Transaktionen, die nur wenige Daten ändern.

  ➔ Schlechte Leistung des Gesamtsystems

- **Lösung**
  - Lesende Transaktionen erhalten für den Zugriff einen Snapshot der Datenbank.
  - Schreibende Transaktionen benutzen die üblichen Lock-Protokolle, wie z. B. MV2L.
- **Problem**
    - Wie erkennt man Transkationen, die nur lesend auf die Daten zugreifen?

Prinzipielle Vorgehensweise

- **Beim Start einer Transkation T werden folgende Schritte ausgeführt.**
  - T benutzt einen eigenen Snapshot der Datenbank.
  - Änderungen werden nur auf dem lokalen Snapshot durchgeführt.
  - Änderungen auf der Datenbank durch andere Transaktionen sind zunächst für T nicht sichtbar.
  - Beim Commit werden die Änderungen in die Datenbank übertragen.

- **Wichtige Regel (First-committer-wins)**
  - Wenn beim Commit Daten geschrieben werden, die bereits von parallel laufenden Transaktionen geschrieben wurden, wird die Transaktion abgebrochen.

Beispiel

Datenbankzustand: x = y = z = 0

T1	T2	T3
w(y,1);		
c ();		
	r(x); // x = 0	
	w(y,2);	
		w(x,2);
		w(z,3);
		c();
	r(z); // z = 0	
	r(y); // y = 2	
	w(x,3);	
	cReq();	
	abort();	

Zeit

Konkurierende Updates sind nicht zu sehen

Eigene Updates sind bekannt

x wurde bereits in T3 geändert,

Problem beim Serialisieren erkannt.

Vorteile von SI

- **Kein Blockieren lesender Transaktionen,**
  - Und diese behindern auch keine anderen Transaktionen.
- **Leistung entspricht dem von dem Isolationslevel Read Committed**
- **Vermeidung der üblichen Anomalien**
  - Keine dirty reads
  - Keine lost updates
  - Keine non-repeatable reads
  - Keine Phantome

- **Problem mit SI**
  - Keine Garantie der strikten Serialisierbarkeit!

Beispiel

- **Im Folgenden betrachten wir Transaktionen der Form:**
  Transaction T (input String)
  1. x ← select count (*) from doctors where on-call = true;
  2. If (x >= 2)

     update doctors
     set on_call = false
     where name = input;

- Transaktionen: T_1 = T (Alice) und T_2 = T(Bob)

- Falls T_1 und T_2 parallel laufen und zunächst beide den ersten Befehl und danach den zweiten ausführen, kann das Ergebnis weder durch die beiden möglichen Abläufe
  - $T_1 – T_2$
  - $T_2 – T_1$

  erzeugt werden.

➔ **Keine Serialisierbarkeit**

Kommerzielle Umsetzungen

- **In Oracle kann der Isolationslevel Serializable nur die SI-Eigenschaft garantieren.**
- **In einigen DBMS, wie z. B. SQL Server, gibt es zusätzlich einen weiteren Isolationslevel Snapshots.**
- **In PostrgeSQL wird die sogenannte Serializable Snapshot Isolation verwendet.**
  - Dabei wird die Serialisierbarkeit garantiert ohne dabei die Laufzeitvorteile von SI zu verlieren.
  - Hierfür muss man beim Starten einer Transaktion noch diese Option wählen.
    - start transaction isolation level serializable;
  - Ohne diese Option wird per Default die Option read commited unterstützt.

Literatur

- H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil (1995): "*A Critique of ANSI SQL Isolation Levels",* SIGMOD Conference.
- A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, D. Shasha (2005): *Making Snapshot Isolation Serializable, ACM TODS.*
- M. Cahill, U. Röhm and A. Fekete (2008): *Serialisable Isolation for Snapshot Databases,* SIGMOD Conference.
- Ports, D. R., & Grittner, K. (2012). Serializable snapshot isolation in PostgreSQL. *Proceedings of the VLDB Endowment*, *5*(12), 1850-1861.

- **Auf diese Artikel kann z. B. über Google Scholar zugegriffen werden.**

5.4 Fehlerbehandlung

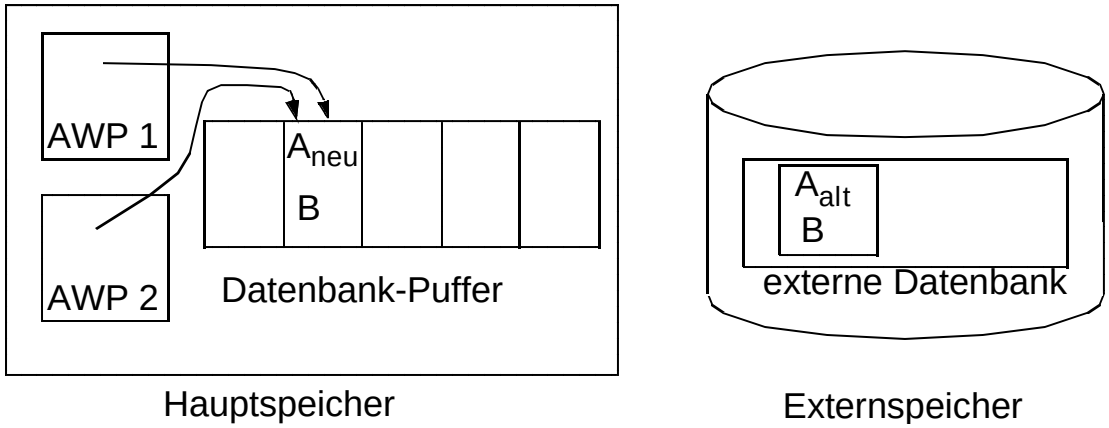
- **Problem**
  - Schutz der Datenbank vor Beeinträchtigungen durch Fehler des Systems oder eines Benutzers
  - Nach Systemabsturz innerhalb einer TA
    - inkonsistenter Zustand der DB
    - physische und logische Inkonsistenz

- **Lösung**
  - Recovery-Komponente eines DBS
    - Wiederherstellen eines korrekten DB-Zustandes
      - basiert auf dem TA-Konzept des DBS

Fehlerklassen

- **Transaktionsfehler** (z. B. Deadlock, Konsistenzverletzung, Division durch 0)
  - ➔ Rücksetzen einer oder mehrerer TAs im laufenden Betrieb
  - ➔ Neustart der Transaktion
- **Systemfehler** (DBS ist funktionsunfähig, Verlust des Inhalts im Hauptspeicher)
  - ➔ Rückgängigmachen aller laufenden TAs
  - ➔ Neustart des Systems
- **Speicherfehler** (Verlust des Plattenspeichers durch “head crash”, selten)
  - ➔ Rekonstruieren der Datenbank

Lesen und Schreiben von Datensätzen



- Datensätze werden in **Seiten** (**Blöcken**) auf dem Externspeicher abgelegt.
  - Eine Seite ist die kleinste Transfereinheit zwischen Extern- und Hauptspeicher.
- Ein DBS besitzt einen **Puffer**, in dem Datenseiten (und ihre Datensätze) für die AWPs bereitgestellt werden.
  - Zu jeder Seite im Puffer gibt es genau eine Seite in der externen DB
    - Seite in der externen DB ist aber ggf. veraltet.
    - Schreiben einer Seite in die DB zerstört den alten Zustand.

Ablauf einer Lese/Schreiboperation

1. **Lese die Seite** vom Externspeicher in den Puffer (wenn nicht bereits vorhanden)
2. **Fixiere die Seite** im Puffer (**Fix**), d.h. die Seite bleibt fest im Hauptspeicher.
3. **Setze eine Sperre** auf den gewünschten Datensatz.

   AWP liest/schreibt den Datensatz und führt weitere Operationen aus.

4. **Zurückgeben der Sperre**.
5. Kennzeichne, dass das AWP die Seite nicht mehr benötigt (**Unfix**).

Gefahr einer inkonsistenen Datenbank

- **Im Fall von Schreiboperationen**
  - Modifizierte Seiten im Puffer werden nicht sofort auf den Externspeicher übertragen.
    - externe DB ist veraltet

      ➔ Datenbank gerät kurzzeitig in einen inkonsistenten Zustand (innerhalb einer TA)
    - externe DB hat nach dem Ende der TA einen inkonsistenten Zustand.

      ➔ Verlust des Hauptspeichers ➔ inkonsistente DB

- **Ziel**
  - DB soll auch bei Verlust des HSP in einem konsistenten Zustand sein.

Varianten beim Lesen/Schreiben

- **Freigabe von Seiten**
  - Pufferverwaltung kann diese Seite aus dem Puffer entfernen und (wenn die Seite geändert wurde) auf den Externspeicher schreiben
    - Varianten
      - **steal**: Freigabe bereits vor dem commit einer TA
      - **no-steal**: keine Freigabe vor dem commit der TA.
- **Schreiben modifizierter Seiten auf den Externspeicher**
    - Varianten
      - **force**: Schreiben der Seiten beim commit
      - **no-force**: Schreiben der Seiten zu einem späteren Zeitpunkt
- **in heutigen DBMS:**
    - **no-force und steal**

Protokollierung von Änderungsoperationen

- Mögliche Fälle nach einem commit einer TA (im Fall von steal und no-force):
  - externe DB ist in einem inkonsistenten Zustand
  - Änderungen sind noch nicht in der externen Datenbank
- Protokoll
  - REDO-Information:
    wenn Änderungen nachvollzogen werden sollen.
  - UNDO-Information:
    wenn Änderungen rückgängig gemacht werden sollen.
- Eintrag in der Protokolldatei (Log) besteht aus:
  - LSN: Log Sequence Number - eindeutige Kennung (monoton wachsend)
  - TA_ID: Transaktionskennung
  - SID: Seitennummer
  - REDO-Information
  - UNDO-Information
  - P_LSN: Zeiger auf den vorherigen Log-Eintrag der Transaktion TA_ID.

Varianten beim Logging

- **Physisches Logging**
  - Explizite Angabe des neuen Zustands (“after image”) und
  - des alten Zustands (“before image”) des Objekts.
- **Logisches Logging**
  - Angabe der Operation (für redo) und der zugehörigen Umkehrfunktion (undo)
- **Physiologisches Logging**
  - physisches Logging auf der Seitenebene
  - logisches Logging innerhalb einer Seite

Logisches Logging

Transaktion T1	Transaktion T2	Logisches Logging
T1.bot()		<#1, T1, bot, 0>
T1.r(A,a_{old})		
	T2.bot()	<#2, T2, bot, 0>
	T2.r(C,c_{old})	
$a_{new} = a_{old} - 10$		
T1.w(A,$a1_{new}$)		<#3, T1, PA, &A, -10, +10, #1>
	$c_{new} = c_{old} + 20$	
	T2.w(C, c_{new})	<#4, T2, PC, &C,+20, - 20, #1>
T1. r(B, b_{old})		
$b_{new} = b_{old} + 10$		
T1. w(B, b_{new})		<#5, T1, PB, &B,+10, -10, #3>
T1.c()		<#6, T1, commit, #5>
	T2.r(A,xold)	
	xnew = xold - 20	
	T2.w(A,xnew)	<#7, T2, PA, &A, -20, +20, #4>
	T2.c()	<#8, T2, commit, #7>

Eintrag für bot

Kennung der Transaktion

Nummer des Log-Eintrags

Eintrag für Schreiben

Kennung der Seite

inverse Operation

Operation

Position in der Seite

Verweis auf den Vorgängereintrag

Eintrag für commit

Physisches Logging

Transaktion T1	Transaktion T2	Logisches Logging
T1.bot()		<#1, T1, bot, 0>
T1.r(A,a_{old})		
	T2.bot()	<#2, T2, bot, 0>
	T2.r(C,c_{old})	
$a_{new} = a_{old} - 10$		
T1.w(A,$a1_{new}$)		<#3, T1, PA, &A, a_{new}, a_{old}, #1>
	$c_{new} = c_{old} + 20$	
	T2.w(C, c_{new})	<#4, T2, PC, &Cc_{new}, c_{old}, #1>
T1. r(B, b_{old})		
$b_{new} = b_{old} + 10$		
T1. w(B, b_{new})		<#5, T1, PB, &B, b_{new}, b_{old}, #3>
T1.c()		<#6, T1, commit, #5>
	T2.r(A,xold)	
	$x_{new} = x_{old} - 20$	
	T2.w(A,xnew)	<#7, T2, PA, &A, x_{new}, x_{old}, #4>
	T2.c()	<#8, T2, commit, #7>

After Image

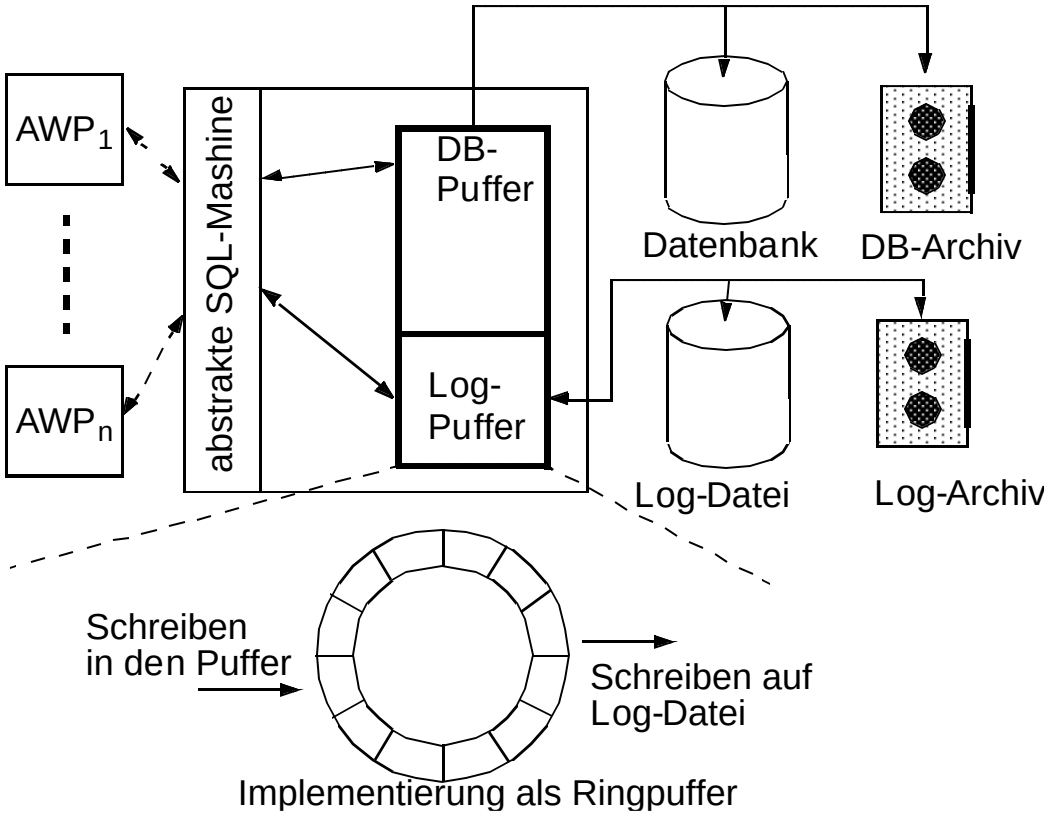
Before Image

Diskussion

- Physisches Logging
  - Unter Verwendung der UNDO- und der REDO-Information ist es also möglich den vorherigen Zustand der Seite zu rekonstruieren.
- Logisches Logging
  - Es muss noch zusätzlich der **Zustand der betroffenen Seite gekennzeichnet werden:**

    ➔ **LSN** der zuletzt auf der Seite wirksamen Schreiboperation wird **in der Seite zusätzlich abgespeichert.**
  - Zwei Fälle
    1. LSN der Seite < LSN eines Protokolleintrag ➔ ?
    2. LSN der Seite >= LSN des Protokolleintrags ➔ ?

Verwaltung der Protokolleinträge



WAL-Prinzip

- WAL = Write Ahead Logging
- Folgende Regeln müssen beim Schreiben der Log-Einträge befolgt werden.
  - Vor dem commit einer TA müssen alle zugehörigen Protokolleinträge in die Log-Datei geschrieben werden.
  - Vor dem Schreiben einer modifizierten Seite in die (externe) Datenbank müssen alle zugehörigen Protokolleinträge geschrieben werden.
  - Wenn ein Protokolleintrag mit LSN x in die Log-Datei geschrieben wird, so müssen vorher alle Einträge mit LSN y, y < x, geschrieben worden sein.
- Vorteil
  - Schreiben von Log-Einträgen ist günstig.

    ➔ Sequentielles Schreiben von Einträgen
  - Konsistenz kann durch Log-Einträge sichergestellt werden.

5.4.1 Wiederanlauf nach einem Systemfehler

- **Ursache**
  - Verlust des Hauptspeichers
- **Zwei Arten von TA**
  - **Winner-Transaktionen**
    - Für TAs, die bereits mit commit beendet wurden, müssen die durchgeführten Änderungen in der DB nachvollzogen werden. **Warum ?**
  - **Loser-Transkationen**
    - Für TAs, die zum Zeitpunkt es Absturzes aktiv waren, aber noch nicht mit commit beendet wurden, müssen die Änderungen rückgängig gemacht werden.
- **Wiederanlauf geschieht in drei Phasen:**
  1. Analyse: Bestimme Winner und Loser
  2. Wiederholung der Historie (REDO)
  3. Zurücksetzen der Loser (UNDO)

Analysephase

- **Ablauf**
  - sequentielles Durchlaufen der Log-Datei vom Anfang bis zum Ende
    - TA mit einem Eintrag “bot” und einem Eintrag “commit” sind **Winner.**
    - TA mit einem Eintrag “bot” ohne einem Eintrag “commit” sind **Loser.**

REDO-Phase

- **Ablauf**
  - Sequentielles Durchlaufen der Einträge in der Log-Datei **vom Anfang bis zum Ende**
    - Lese die zugehörige Seite vom Externspeicher
      - Falls **LSN der Seite < LSN des Eintrags**:
        (i) Führe REDO-Operation aus.
        (ii) Übertrage die LSN des Eintrags in die Seite.

UNDO-Phase

- **Ablauf**
  - Sequentielles Durchlaufen der Einträge in der Log-Datei **vom Ende bis zum Anfang**
    - Führe für jede **Loser-TA** die UNDO-Operation aus.
      - Lesen der Seite, modifizieren des Eintrags und Schreiben der Seite.
  - Zusätzlich:
    - Schreiben von **Kompensationseinträgen in die Log-Datei**

Schicksalsstunden im Leben einer Datenbank

- **Folgender Fall kann passieren (und ist natürlich schon zu oft passiert):**
  - Stromausfall: Verlust es Hauptspeichers
  - Wiederanlauf des Systems
  - erneuter Stromausfall (**vor dem Ende des Wiederanlaufs**)

- **Anforderung**
  - **Idempotenz der UNDO- und REDO-Operationen**
    - Ergebnis einer beliebig oft ausgeführten UNDO/REDO-Operation entspricht dem Ergebnis einer einmalig ausgeführten UNDO/REDO-Operation
  - Offensichtlich gilt:
    - REDO-Operationen sind idempotent

Idempotenz der UNDO-Operation

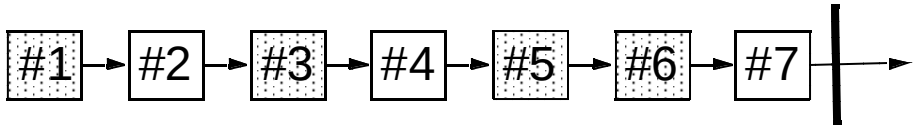
- Diese Eigenschaft kann durch **Kompensationseinträge** in der Protokolldatei sichergestellt werden.

- Für jede ausgeführte UNDO-Operation wird ein Eintrag in die Log-Datei geschrieben:
  - Dieser besitzt keine UNDO-Information.
  - Seine REDO-Information entspricht dabei der (ausgeführten) UNDO-Operation.
  - Zusätzlich gibt es noch einen Verweis auf einen Eintrag in der Log-Datei (UNDO_P_LSN).
    - UNDO_P_LSN ist der Vorgängereintrag der zuletzt ausgeführten UNDO-Operation in der Log-Datei

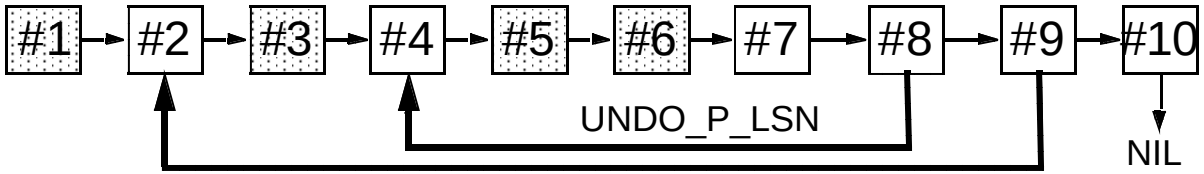
Beispiel

- **Beispiel Transkationen T1 und T2**

Log-Datei (vor dem Wiederanlauf)



Log-Datei nach dem Wiederanlauf



5.4.2 Zurücksetzen einer TA

- **Gründe für das Zurücksetzen einer TA**
  - System muss eine oder mehrere TAs zurücksetzen (z. B. wegen einer Verklemmung).
  - Benutzer bricht seine TA ab.
- **Anforderung:**
  - Alle DB-Änderungen der TA müssen zurückgenommen werden.
  - Lokal für eine TA möglich, wenn noch keine Sperren freigegeben wurden. → Striktes 2PL
- **Ablauf**
  - Sequentielles Durchlaufen der Protokolldatei **vom Ende** bis zum ersten Eintrag der TA, die zurückgesetzt werden soll:
    1. Ausführen der UNDO-Operation
    2. Schreiben eines Kompensationseintrags
    3. Aufsuchen des nächsten Eintrags (mit P_LSN)
  - Von der TA gehaltene Sperren müssen beim Rücksetzen freigegeben werden.

Sicherungspunkte

- Nachteil beim Zurücksetzen: Protokolldateien können **SEHR GROSS** sein!
  - ➔ Recovery wird sehr teuer!
- Einführung von **Sicherungspunkten**, so dass
  - der Wiederanlauf startet am letzten Sicherungspunkt
  - und ältere Protokolleinträge können gelöscht werden.

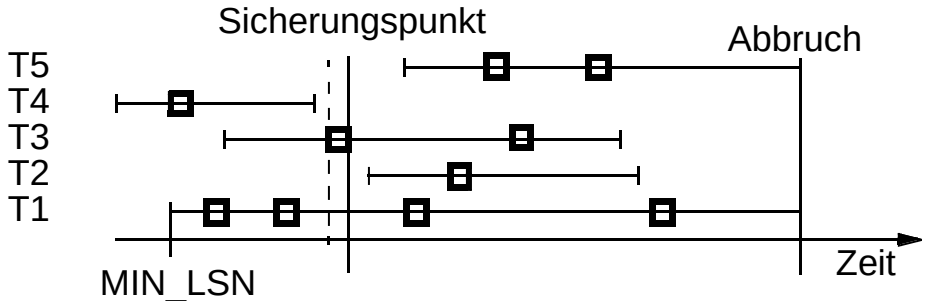
- **Arten von Sicherungspunkten (Engl.: savepoints, checkpoints)**
  - Transaktionskonsistente Sicherungspunkte
  - Aktionsbasierte Sicherungspunkte
  - Unscharfe Sicherungspunkte

Transaktionskonsistente Sicherungspunkte

- **Erzeugung des Sicherungspunkts**
  - Überführung des Systems in einen Ruhezustand (➔ keine TA ist aktiv)
    - Schreiben aller modifizierten Seiten im Puffer
    - Neuinitialisierung der Protokolldatei
    - Starten der wartenden TA
- **Nachteile**
  - Ruhestand eines Systems ist nicht akzeptabel
    - zu lange Verzögerung der wartenden TA
  - Großer Aufwand beim Schreiben des Puffers
    - Große Puffer können 1.000.000 Seiten (= 40 GB) und mehr besitzen.

Aktionsbasierte Sicherungspunkte

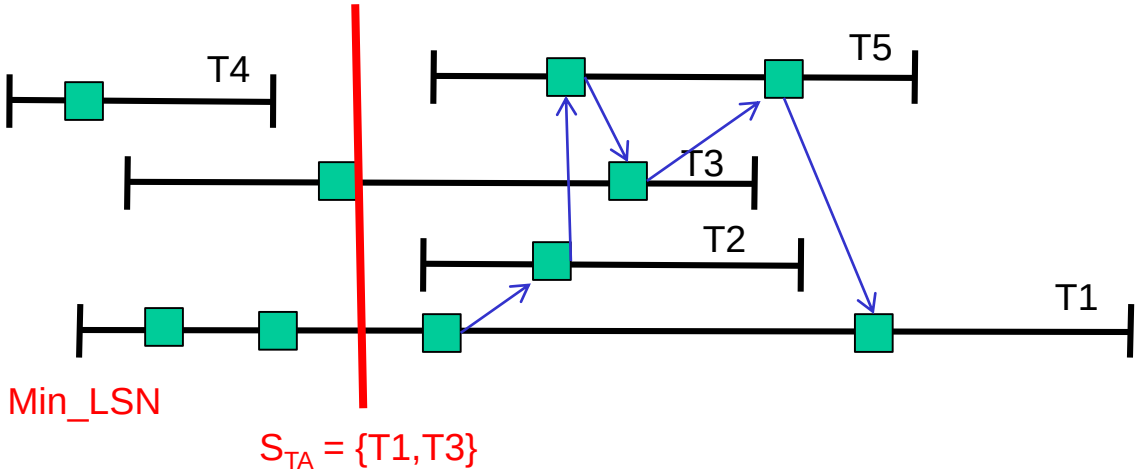
- Idee
  - Keine Beruhigung des Systems erforderlich.
  - Stattdessen müssen nur die elementaren Änderungsoperationen abgeschlossen werden



- Ablauf
  - Schreiben des Log-Puffers und des DB-Puffers (WAL-Prinzip !)
  - Berechnen der **Liste S_{TA}** aller zum Zeitpunkt des Sicherungspunktes aktiven TAs
  - Berechnen und Speichern von **MIN_LSN** = min {LSN | LSN gehört zu einer TA aus S_{TA}}
- Auswirkungen für die Recovery
  - Analyse- und REDO-Phase setzt beim Sicherungspunkt auf.
  - UNDO-Phase muss aber bis MIN_LSN gehen.

Unscharfe Sicherungspunkte

- **Problem bei aktionsbasierten Sicherungspunkten**
  - zu hoher Aufwand beim Schreiben des Puffers
- **Idee**
  - Schrittweises Weitersetzen von Sicherungspunkten



Erforderliche Datenstrukturen

- Liste mit “schmutzigen” Pufferseiten
  - Reihenfolge bzgl. der LSN der zuletzt in der Seite ablaufenden Änderung.
  - Seite am Kopf der Liste besitzt die kleinste LSN (MIN_LSN_DIRTY).
  - MIN_LSN_DIRTY ist quasi die LSN des Sicherungspunkts
- Menge S_{TA}
  - alle zum Zeitpunkt MIN_LSN_Dirty aktiven TAs
- MIN_LSN
  - kleinste LSN, die in TAs aus S_{TA} vorkommt.

5.4.3 Verlust des externen Speichermediums

- **Idee**
  - Verwendung von Archivkopien für Datenbanken und Log-Dateien
    - transaktionssicherer Datenbankzustand
    - Schreiben der Log-Daten immer ins Archiv
    - ➔ Bei Verlust der Datenbank ist dadurch die Herstellung des konsistenten Zustands der Datenbank sichergestellt.

Zusammenfassung

- **Transaktionskonzept in Datenbanken**
  - Sicherstellung der ACID-Eigenschaften
  - Serialisierbarkeit von Transaktionen
    - Konflikte
  - Synchronisation von TAs
    - Striktes 2PL
- **Recovery-Komponente**
  - Sicherstellung der Eigenschaften Dauerhaftigkeit und Konsistenz
  - Fehlerklassen
  - WAL-Prinzip und Log-Dateien
  - Sicherungspunkte