# TRANSAKTIONEN UND DATENINTEGRITÄT

„Concurrency Control and Recovery in Database Systems“

P.A. Bernstein, V. Hadzilacos, N. Goodman

**Addison Wesley, 1987.**

Kapitel 1. und 6.

Begriffe:

## Integritätsbedingungen und Konsistenz:

Eine Datenbank ist konsistent, wenn sie eine Menge vorgegebener logischer Bedingungen erfüllt, die Integritätsbedingungen genannt werden.

# Transaktionen:

Ein Benutzer verwendet eine Datenbank, indem er eine Folge von Zugriffsbefehlen (Lese- und Update-Befehle) auf dieser Datenbank ausführt.

Eine derartige Befehlsfolge wird Transaktion genannt und führt die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand über.

Führt eine Transaktion nur Lesezugriffe durch, dann wird sie als Lese-Transaktion bezeichnet, sonst als Update-Transaktion.

# Eigenschaften von Transaktionen (ACID):

1. ***Atomizität* (atomicity)**
   Ausführung entweder zur Gänze oder gar nicht.

2. ***Konsistenz* (consistency)**
   Eine Transaktion führt von einem konsistenten Zustand in einen anderen konsistenten Zustand.

3. ***Isolation***
   Teilergebnisse bleiben bis zum Ende einer Transaktion für alle anderen unsichtbar.

4. ***Dauerhaftigkeit (durability)***
   Die Ergebnisse einer erfolgreich durchgeführten Transaktion bleiben erhalten.

**Beispiel:** Bank-Überweisung von Konto auf Sparbuch.

Datenstruktur:

```
SPARBUCH (SPNR, EINLAGE, BESITZER)
KONTO (KNR, KSTAND)
SPAREINLAGEN-GESAMT
```

Integritätsbedingung:

```
KSTAND >= 0
EINLAGE > 0
SPAREINLAGE-GESAMT = Summe EINLAGE
```

## Transaktion: Umbuchung

Buche 500,- € von Konto 123 auf Sparbuch 321

```
begin transaction
update KONTO
 set KSTAND = KSTAND - 500
 where KNR = 123;
update SPARBUCH
 set EINLAGE = EINLAGE + 500
 where SPNR = 321;
SPAREINLAGE-GESAMT :=
 SPAREINLAGE-GESAMT + 500
end transaction
```

# Parallelausführung von Transaktionen (concurrent execution of transactions)

**Vereinfachtes Modell**:

atomare Objekte:

haben **Name** und **Wert**

atomare Datenbank-Operationen:

Wert lesen: Read(x)

Wert ändern: Write(x)

Write(x, Wert)

Transaktions-Operationen:

Start, Commit, Abort

## Beispiel:

Transaktion 1:

Überweisung von Konto auf Sparbuch:

```
Read(Konto)
Konto := Konto - 1.000
Write(Konto)
Read(Sparbuch)
Sparbuch := Sparbuch + 1.000
Write(Sparbuch)
```

Transaktion 2:

Überweisung der Telefonrechnung:

```
Read(Konto)
Konto := Konto - 174,38
Write(Konto)
Read(Tele-Konto)
Tele-Konto := Tele-Konto + 174,38
Write(Tele-Konto)
```

# Transaktions-Operationen

## Start

Die Operation Start kennzeichnet den Beginn der Ausführung einer neuen Transaktion.

## Commit

Durch das Ausführen einer Commit Operation teilt ein Programm der Datenbank mit, daß die Transaktion normal beendet wurde und alle Änderungen permanent gemacht werden sollen.

## Abort

Durch Ausführen einer Abort Operation teilt ein Programm der Datenbank mit, daß die Transaktion abnormal beendet wurde und alle Änderungen verworfen werden sollen.

# Transaktions Syntax

Benutzersicht: Eine Transaktion ist die Ausführung von einem oder mehreren Programmen die Datenbank- und Transaktionsoperationen beinhalten.

Beispiel: Banking Datenbank

Transaktion: Buche Geld von einem Konto auf ein anderes.

```
Procedure Transfer begin
 Start;
 input(fromaccount, toaccount,
        amount);
 temp := Read(Accounts
              [fromaccount]);
 if temp < amount then begin
    output("Kontostand nicht
              ausreichend");
    Abort;
 end
 else begin
    Write(Accounts[fromaccount],
       temp - amount);
    temp := Read(Accounts
                 [toaccount])
    Write(Accounts[toaccount],
       temp + amount);
    Commit;
    output("Buchung vollständig");
 end;
 return;
end
```

Die dafür verwendete Sprache kann sein:

- eine Datenbank-Abfragesprache (Query Language)
- eine Report-Beschreibungssprache (4GL)
- Eine High-Level Programmiersprache mit eingebetteten Datenbank Operationen.

# Commit und Abort

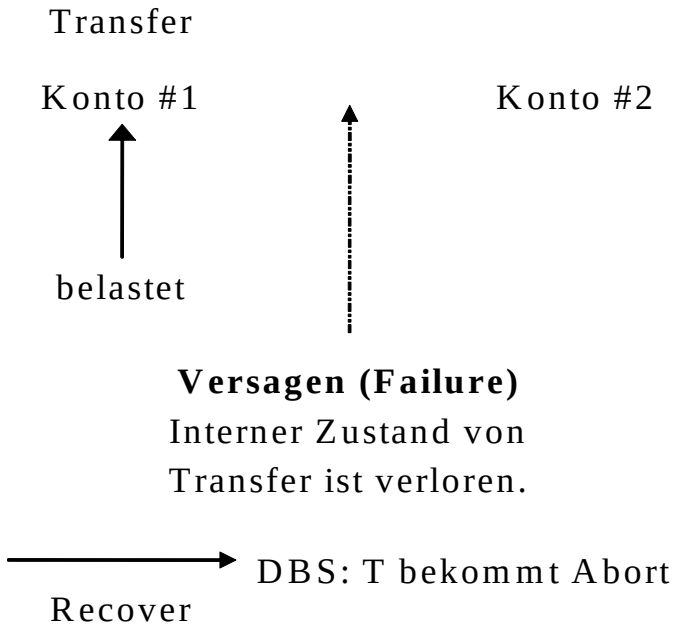
Eine Transaktion die eine

- **Start** Operation ausgeführt hat, heißt **aktiv**.
- **Commit** Operation ausgeführt hat, heißt **committed**.
- **Abort** Operation ausgeführt hat, heißt **aborted**.

Eine Transaktion heißt **uncommitted** wenn sie aktiv oder aborted ist.

Eine Abort Operation kann

- durch die Transaktion ausgeführt werden, oder
- durch die Datenbank verursacht werden (Systemversagen etc.).



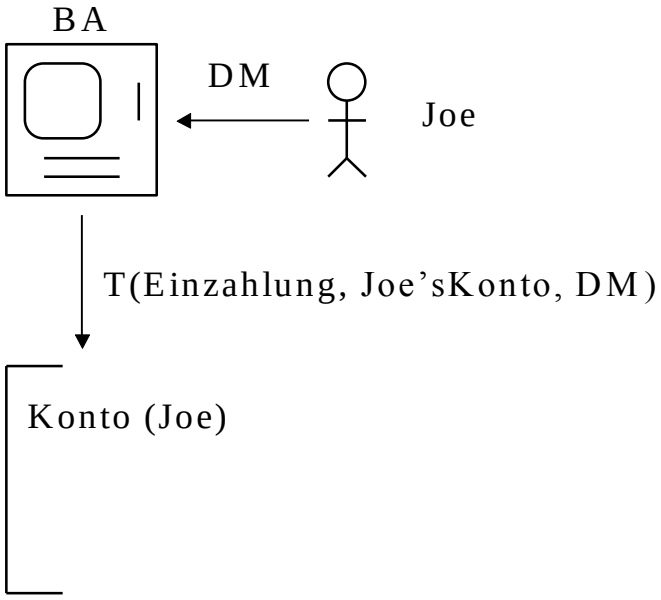

Der Abort wird als Operation der Transaktion betrachtet, auch wenn tatsächlich die Datenbank den Abort ausgelöst hat.

- Wenn eine Transaktion aborted wird, muß die Datenbank alle Auswirkungen der Transaktion verwerfen.

- Es wird die Fähigkeit benötigt, einen Punkt in der Zeit festzulegen, nach dem die Datenbank garantiert, daß die Transaktion nicht aborted wird und alle ihre Auswirkungen permanent werden.

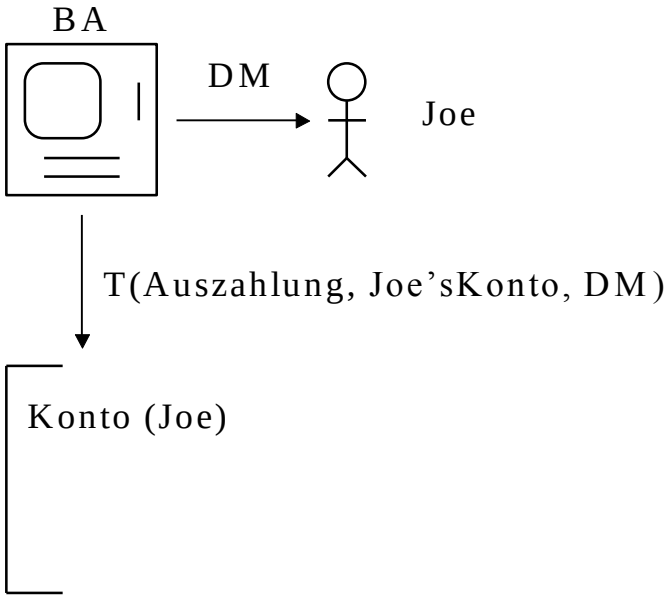
**Beispiel:** Bankautomat (ATM)

Aus Sicht des Benutzers:




Der Benutzer will den Bankautomaten nicht verlassen, bevor er sicher ist, daß die Einzahlung nicht aborted wird.

Aus Sicht der Bank:



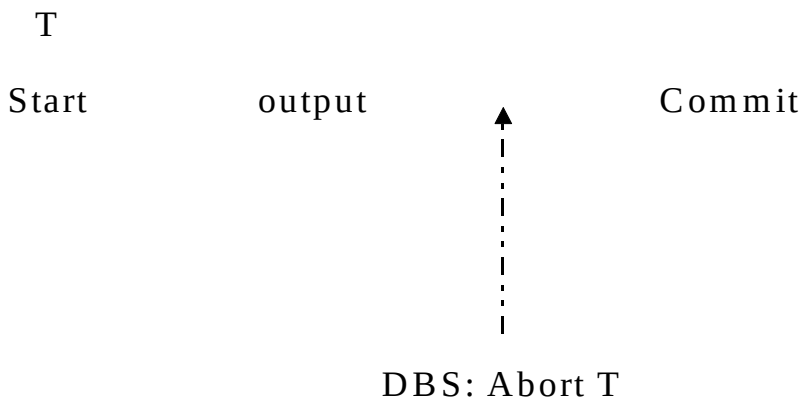

Bei der Verarbeitung einer Auszahlung soll der Bankautomat kein Geld ausgeben, bevor sichergestellt ist, dass die Abbuchungs-Transaktion nicht aborted wird.

## Commit Operation:

Die Ausführung einer Commit Operation drückt aus, dass eine Transaktion "normal" beendet wurde und alle Auswirkungen permanent gemacht werden sollten.

Die Ausführung eines Commit ist eine Garantie durch das DBS, dass es die Transaktion nicht aborten wird und alle Auswirkungen der Transaktion zukünftige Fehler des Systems überstehen.

## Probleme:




Benutzer: Vertraue der Ausgabe erst, wenn die Datenbank mitteilt, dass T committed hat.

# Recoverability

Das Recovery System soll die Datenbank in einen Zustand bringen, in dem die Datenbank sich verhält, als ob sie alle Auswirkungen von committed Transaktionen enthält und keine der Auswirkungen von uncommitted Transaktionen.

Abort T

Das DBS muß alle Auswirkungen von T zurücksetzen:

- Auswirkungen auf Daten ( Write(x) )
- Auswirkungen auf andere Transaktionen ( T': Read(x) )

## Beispiel:

- Initiale Wert von x und y sind "1"
- Zwei Transaktionen $T_1$ und $T_2$

$$Write_1(x,2);\ Read_2(x);\ Write_2(y,3)$$

Annahme: $T_1$ aborted.

Das DBS verwirft die Operation $Write_1(x,2)$ und setzt x damit auf den ursprünglichen Wert 1.

Da aber $T_2$ den Wert von x gelesen hat der durch $T_1$ geschrieben wurde, muss $T_2$ ebenfalls zurück-gesetzt werden (Cascading Abort), der Wert von y wird deshalb ebenfalls auf 1 zurückgesetzt.

# Recoverability

Recoverabilty sichert zu, dass das abbrechen (aborting) einer Transaktion nicht zu einer veränderten Semantik bei committed Transaktionen führt.

## Beispiel:

x = y = 1

```
T1 Write(x,2)
T2 Read(x)      /* T2 Reads from T1 */
T2 Write(y,3)
T2 Commit
```

Das Commit von $T_2$ folgt nicht dem Commit von $T_1$

→ **Non Recoverable Execution!**

Eine Transaktion $T_j$ **liest von** (reads from) einer Transaktion $T_i$, wenn:

1. $T_j$ liest x nachdem $T_i$ auf x geschrieben hat;

2. $T_i$ bricht nicht ab bevor $T_j$ x liest;

3. Jede Transaktion (sofern vorhanden) die x schreibt, nachdem $T_i$ x geschrieben hat und bevor $T_j$ x liest, bricht ab bevor $T_j$ x liest.

- Eine Ausführung ist **Recoverable**, wenn für jede Transaktion T die ein Commit ausführt, T's Commit dem Commit jeder anderen Transaktion folgt, von der T liest.

- T kann nicht Committen, bevor alle Transaktion, von denen T liest, garantiert nicht abbrechen (also committed sind).

→ recoverble Ausführung

$T_1$ Write(x,2)
$T_2$ Read(x)
$T_2$ Write(y,3)
$T_2$ Commit

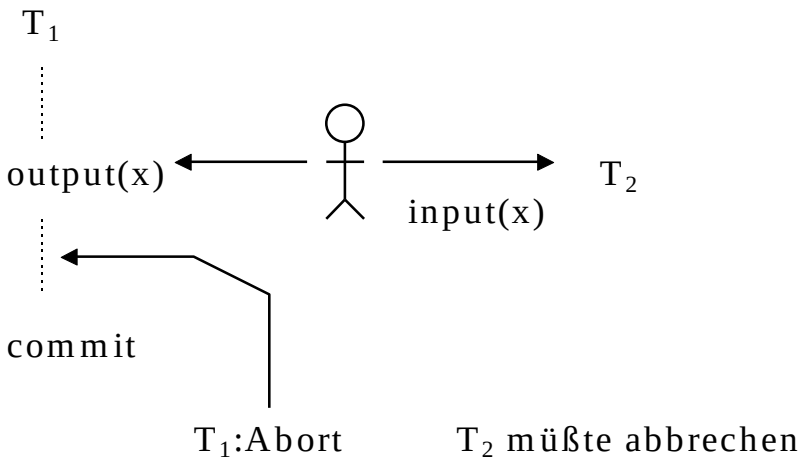
Das DBS muss das Commit von $T_2$ verzögern, dann kann $T_2$ abbrechen, wenn $T_1$ abbricht.

→ Verzögern eines Commit ist eine Methode, um sicherzustellen, dass Ausführungen recoverable sind.

# Terminal I/O

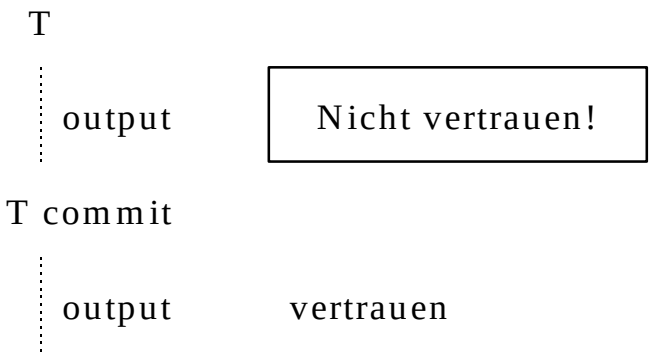
Input und Output Operationen sind eine andere Methode mit der Transaktionen indirekt miteinander kommunizieren können.

**Beispiel:**




Aber: Das DBS weiß nichts über diesen Zusammenhang!

Es handelt sich um ein Benutzerproblem!




→ Das DBS kann T's Ausgabe verzögern bis T committed.

- normalerweise O.K., aber
- Funktioniert nicht immer.

```
T
:
:
output
input
:
:
T commit
```

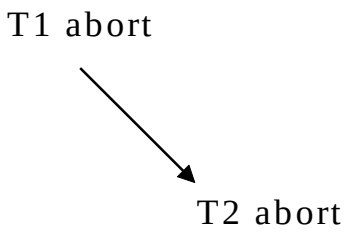
# Vermeidung von Cascading Aborts

```
x = y = 1

T1: Write(x,2)
T2: Read(x)
T2: Write(y,3)
T1: Abort
```

T2 muss ebenfalls abgebrochen werden!

Kaskadierter Abbruch (cascaded abort):




Nicht sehr schön, weil:

- schwerwiegend
- unkontrolliert!

DBS sind designed um kaskadierte Abbrüche zu vermeiden

Ein DBS vermeidet kaskadierte Abbrüche, indem es sicherstellt, dass jede Transaktion nur Werte liest die von bereits committed Transaktionen geschrieben wurden.

→ Das DBS verzögert jedes Read(x), bis alle Transaktionen die vorher ein Write(x,val) ausgeführt haben aborted oder comitted sind.

# Strikte Ausführung

Aus einer praktischen Sicht heraus ist das Vermeiden von kaskadierten Abbrüchen nicht immer Ausreichend.

**Beispiel:**

```
x = y = 1

T1: Write(x,1)
T1: Write(y,3)
T2: Write(y,1)
T1: Commit
T2: Read(x)
T2: Abort
```

- T2 muss zurückgesetzt werden
- kein kaskadierter Abbruch
- Alle Operationen von T2 werden gelöscht.

Ergebnis:

```
T1: Write(x,1)
T1: Write(y,3)
T1: Commit

y = 3 !!
```

Before Image y von „T2: Write(y,1)“ ist 3!

Das DBS implementiert Aborts durch Wiederherstellung der Before Images aller Writes einer Transaktion.

→ Dies ist allerdings nicht immer korrekt!

**Beispiel:**

```
x = 1

T1: Write(x,2)
T2: Write(x,3)
T1: Abort
```

- Das Before Image von „T1: Write(x,2)“ ist 1
- Der Wert von x sollte aber auf 3 gesetzt werden!

Beispiel (Fortsetzung):

```
x = 1

T1: Write(x,2)
T2: Write(x,3)
T1: Abort
T2: Abort
```

- Das Before Image von „T2: Write(x,3)“ ist 2
- Der Wert von x sollte aber auf 1 gesetzt werden!
- Das Before Image wurde von einer Aborted Transaction geschrieben.

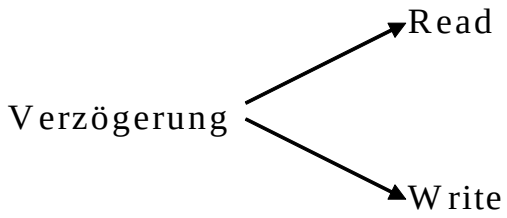
Diskrepanzen zwischen

- Den Werten die (zurück-) gesetzt werden sollen, und den

- Before Images der Writes

→ Die Ausführung eines Write(x,val) sollte solange verzögert werden, bis alle Transaktionen die vorher x geschrieben haben, entweder committed oder aborted sind.

**Strikte Ausführung (strict execution)**




→ No cascading aborts.
Recoverable.

# Serialisierbarkeit (Serializability)

Wenn zwei (oder mehr) Transaktionen gleichzeitig (concurrent) ausgeführt werden, bedeutet dies, dass ihre Datenbankoperationen abwechselnd (interleaved) durchgeführt werden. Dieses Mischen von Operationen verschiedener Transaktionen kann dazu führen, daß sich die Transaktionen gegenseitig beeinflussen.

Dies kann selbst dann geschehen, wenn die Transaktionen selbst korrekt sind und kein Systemversagen auftritt.

**Beispiel:**

```
Procedure Deposit begin
  Start;
  input(account#, amount);
  temp := Read(Accounts[accoount#]);
  temp := temp + amount;
  Write(Accounts[account#],temp);
  Commit
end
```

Annahme:

- Konto 13 hat 1000 €
- Kunde 1 zahlt 100 € ein.
- Kunde 2 zahlt 100.000 € ein
- Beide Kunden rufen das Programm Deposit auf.

Die gleichzeitige Ausführung des Programms Deposit führt zu einer Folge von Reads und Writes auf der Datenbank:

$Read_1$(Accounts[13])
$Read_2$(Accounts[13])
$Write_2$(Accounts[13],101.000)
$Commit_2$
$Write_1$(Accounts[13],1100)
$Commit_1$

Als Ergebnis dieser Ausführung enthält Konto 13 den Wert 1.100 € !

Obwohl Kunde 2's Einzahlung erfolgreich (!) behandelt wurde, wird sie durch die Transaktion von Kunde 1 überschrieben.

Dieses **Lost Update** Problem tritt immer dann auf, wenn zwei Transaktionen, die ein Objekt verändern wollen, das Objekt lesen, bevor eine von beiden es wieder geschrieben hat.

Ein weiteres Problem:

**Gegeben:**

```
Procedure PrintSum begin
  Start;
  input(account1,account2);
  temp1 := Read(Accounts[account1]);
  output(temp1);
  temp2 := Read(Accounts[account2]);
  output(temp2);
  temp1 := temp1 + temp2;
  output(temp1);
  Commit;
end
```

**Annahme:**

- Konto 7 und 86 haben einen Stand von 200 €.
- Kunde 3 führt das Programm PrintSum für die Kontos 7 und 86 aus.
- Kunde 4 transferiert gleichzeitig 100 € von Konto 7 zu Konto 86 (Programm Transfer)

Die gleichzeitige Ausführung dieser beiden Transaktionen führt zu einer Folge von Reads und Writes auf der Datenbank:

$Read_4$(Accounts[7])
$Write_4$(Accounts[7],100)
$Read_3$(Accounts[7])
$Read_3$(Accounts[86])
$Read_4$(Accounts[86])
$Write_4$(Accounts[86],300)
$Commit_4$
$Commit_3$

Das Programm Transfer beeinflusst das Programm PrintSum in diesem Fall so, daß PrintSum 300$ ausgibt, was aber nicht die korrekte Summe der beiden Konten ist.

Die Datenbank befindet sich trotz dieses Fehlers in einem konsistenten Zustand.

Dieses Problem wird **Inconsistent Retrieval** genannt.

## Serielle Ausführungen

Die besprochenen Fehlersituationen treten durch die abwechselnde Ausführung von Operationen verschiedener Transaktionen auf.

Der Benutzer erwartet, dass seine Transaktion nicht beeinflusst wird und so abläuft, als ob sie **alleine ausgeführt** wird.

Der korrekte Ablauf zweier Transaktionen ist, wenn die beiden Transaktionen unabhängig voneinander und **seriell hintereinander** ausgeführt werden.

# Serialisierbare Ausführungen

Eine Ausführung (Schedule, Execution) wird als **serialisierbar** bezeichnet, wenn es eine äquivalente serielle Ausführung gibt.

Zwei Ausführungen sind **äquivalent**, wenn die Auswirkungen auf die Datenbank gleich sind.

Die im Beispiel für Inconsistent Retrieval angegebene Ausführung ist also nicht serialisierbar, da das Ergebnis nicht mit dem einer seriellen Ausführung übereinstimmt.

**Beispiel:** (serialisierbare Ausführung)

$Read_4(Accounts[7])$
$Write_4(Accounts[7],100)$
$Read_3(Accounts[7])$
$Read_4(Accounts[86])$
$Write_4(Accounts[86],300)$
$Commit_4$
$Read_3(Accounts[86])$
$Commit_3$

Diese Ausführung hat das gleiche Ergebnis, wie die serielle Ausführung von Transfer und PrintSum.

Es gibt also serielle, serialisierbare und nicht-serialisierbare Ausführungen von Transaktionen.

## **Beispiel**:

**T1:** READ A; A:=A-10; WRITE A; READ B; B:= B+10; WRITE B;

**T2:** READ B; B:=B-20; WRITE B; READ C; C:=C+20; WRITE C;

| T1 | T2 |
|---|---|
| READ A | |
| A:=A-10 | |
| WRITE A | |
| READ B | |
| B:= B+10 | |
| WRITE B | |
| | READ B |
| | B:=B-20 |
| | WRITE B |
| | READ C |
| | C:=C+20 |
| | WRITE C |

a) serial

| T1 | T2 |
|---|---|
| READ A | |
| | READ B |
| A:=A-10 | |
| | B:=B-20 |
| WRITE A | |
| | WRITE B |
| READ B | |
| | READ C |
| B:= B+10 | |
| | C:=C+20 |
| WRITE B | |
| | WRITE C |

b) serializable

| T1 | T2 |
|---|---|
| READ A | |
| A:=A-10 | |
| | READ B |
| WRITE A | |
| | B:=B-20 |
| READ B | |
| | WRITE B |
| B:= B+10 | |
| | READ C |
| WRITE B | |
| | C:=C+20 |
| | WRITE C |

c) non serializable

**Die Summe ist fest: A+B+C**

# Sperren (Locks)

Wir betrachten die Datenbank als eine Menge von Objekten, die Teile der Datenbank darstellen, die **gesperrt (locked)** werden können.

Durch das Sperren eines Objektes kann eine Transaktion verhindern, dass andere Transaktionen auf dieses Objekt zugreifen.

Objekte können hierbei Attribute, Tupel, Pages, Mengen von Tupeln, ganze Relationen etc. sein.

Welche Granularität für den Lock gewählt werden sollte kann von Anwendung zu Anwendung verschieden sein.

Wir nehmen an, dass wenn ein Teil eines Objektes verändert wird, das ganze Objekt als verändert gilt und einen Wert bekommt, der von anderen Werten unterscheidbar ist.

Generell ist es schwierig, zu testen, ob zwei Ausführungen das gleiche Ergebnis für alle möglichen Startwerte haben, wenn willkürliche Operationen erlaubt sind.

**Praktische Vereinfachung:**

- Werte können nicht gleich sein, wenn sie nicht durch dieselbe Sequenz von Operationen erzeugt wurden.

**Beispiel**:

(A + 10) - 20

hat **nicht** denselben Wert wie

(A + 20) - 30

Dies hat Auswirkungen auf die Äquivalenz von Ausführungen und damit auf unsere Definition der Serialisierbarkeit:

- Wir nennen eine Ausführung nicht-serialisierbar, obwohl sie serialisierbar ist, aber
- Wir nennen niemals eine Ausführung serialisierbar, wenn sie es nicht ist.

Betrachten wir nochmals folgendes Beispiel:

Zwei Transaktionen T1 und T2 führen das Programm P aus:

```
P : READ A; A := A + 1; WRITE A
```

Dies führt zu folgendem Schedule:

| A in Database | 5 | 5 | 5 | 5 | 6 | 6 |
|---|---|---|---|---|---|---|
| T1 : | READ A | | A:=A+1 | | | WRTIE A |
| T2: | | READ B | | A:=A+ 1 | WRITE A | |
| A in T1´s workspace | 5 | 5 | 6 | 6 | 6 | 6 |
| A in T2´s workspace | | 5 | 5 | 6 | 6 | |

Transactions exhibiting a need to lock item A.

Eine Lösung für dieses Problem ist es, einen Lock auf A zu setzen, dafür gelten folgende Regeln:

- Bevor eine Transaktion auf ein Objekt zugreift, muss sie das Objekt sperren (lock).
- Erst wenn eine Transaktion mit der Bearbeitung eines Objektes fertig ist, kann sie es wieder entsperren (unlock).
- In der Zwischenzeit kann keine andere Transaktion auf dieses Objekt zugreifen.
- Versucht eine weitere Transaktion, einen Lock auf das selbe Objekt zu setzen, wird dies verzögert, bis die erste Transaktion das Objekt wieder freigibt.

Eine Ausführung (Schedule) die obigen Bedingungen über Locks genügt heißt **legal.**

Schreiben wir nun das Programm P so, dass es das benutzte Objekt A sperrt:

```
P :  LOCK A; READ A; A := A + 1;
     WRITE A; UNLOCK A
```

Seien wieder T1 und T2 Ausführungen von P.

- Wenn T1 beginnt, versucht sie A zu sperren, sofern keine andere Transaktion A gesperrt hat, wird das System diese Sperrung zulassen.
- Jetzt kann nur noch T1 auf A zugreifen.
- Wenn T2 startet, bevor T1 beendet ist, versucht sie die Anweisung `LOCK A` auszuführen, dies wird dann verzögert, bis T1 A entsperrt.

Die Transaktionen laufen in diesem Fall seriell nacheinander ab!

# Protokolle und Scheduler

**Frage:** Wie können nicht-serialisierbare Ausführungen vermieden werden?

- Scheduler
- Protokolle (die Serialisierbarkeit garantieren)

# Einfaches Modell für Transaktionen

Jede Transaktion $T_j$ ist eine Sequenz von LOCK und UNLOCK Operationen auf Objekten der Datenbank.

$T_j$

```
    LOCK A
  :
  :    /* Tj hält einen Lock auf A */

    UNLOCK A
```

Annahmen:

1. $T_j$ sperrt A nicht, wenn sie bereits einen Lock auf A hält.

2. $T_j$ entsperrt A nicht, wenn sie keinen Lock auf A hält.

3. Immer wenn eine Transaktion einen Lock auf ein Objekt A hält, wird sie den Wert von A verändern, und der Wert, den A nach dem Unlock hat, ist eindeutig und unterscheidbar.

| | | |
|---|---|---|
| $A = V_1$ | $\neq$ | $A = V_2$ |
| LOCK A | | LOCK A |
| UNLOCK A | | UNLOCK A |
| $A = X$ | $\neq$ | $A = X'$ |

# Formelles Modell für das Verhalten von $T_j$

Für alle Sequenzen von LOCK und UNLOCK assoziiere eine Funktion *f()*.

Wenn $A_0$ der Initialwert von A ist, dann ist

$$f_1 f_2 \ldots f_n ( A_0 )$$

der Wert, den A annehmen wird.

Unterscheidbare Werte sind hierbei nicht identisch. Werte werden als uninterpretierte Formeln gesehen.

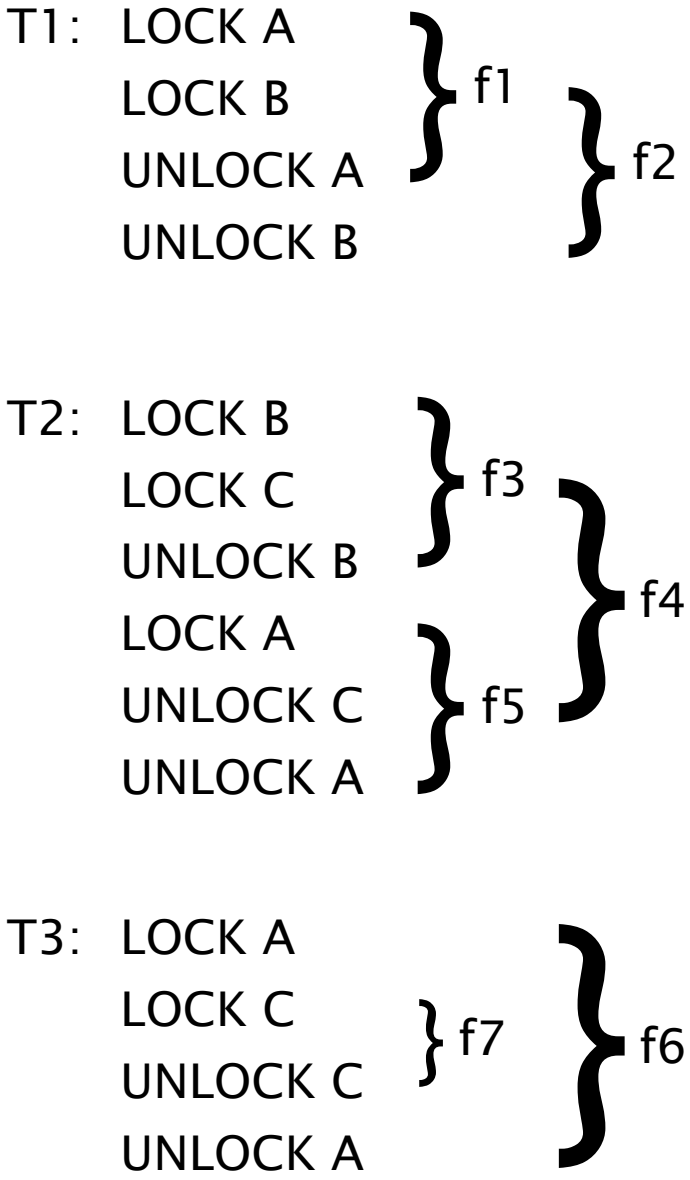
Beispiel:

(A+10)-20 ist **nicht identisch** zu (A+20)-30

Es wird nicht überprüft ob $f_2 f_3 = f_3 f_2$

Beispiel:

Drei Transaktionen T1 T2 T3



| | Step | $A$ | $B$ | $C$ |
|---|---|---|---|---|
| (1) | $T_1$: LOCK $A$ | $A_0$ | $B_0$ | $C_0$ |
| (2) | $T_2$: LOCK $B$ | $A_0$ | $B_0$ | $C_0$ |
| (3) | $T_2$: LOCK $C$ | $A_0$ | $B_0$ | $C_0$ |
| (4) | $T_2$: UNLOCK $B$ | $A_0$ | $f_3(B_0)$ | $C_0$ |
| (5) | $T_1$: LOCK $B$ | $A_0$ | $f_3(B_0)$ | $C_0$ |
| (6) | $T_1$: UNLOCK $A$ | $f_1(A_0)$ | $f_3(B_0)$ | $C_0$ |
| (7) | $T_2$: LOCK $A$ | $f_1(A_0)$ | $f_3(B_0)$ | $C_0$ |
| (8) | $T_2$: UNLOCK $C$ | $f_1(A_0)$ | $f_3(B_0)$ | $f_4(C_0)$ |
| (9) | $T_2$: UNLOCK $A$ | $f_5(f_1(A_0))$ | $f_3(B_0)$ | $f_4(C_0)$ |
| (10) | $T_3$: LOCK $A$ | $f_5(f_1(A_0))$ | $f_3(B_0)$ | $f_4(C_0)$ |
| (11) | $T_3$: LOCK $C$ | $f_5(f_1(A_0))$ | $f_3(B_0)$ | $f_4(C_0)$ |
| (12) | $T_1$: UNLOCK $B$ | $f_5(f_1(A_0))$ | $f_2(f_3(B_0))$ | $f_4(C_0)$ |
| (13) | $T_3$: UNLOCK $C$ | $f_5(f_1(A_0))$ | $f_2(f_3(B_0))$ | $f_7(f_4(C_0))$ |
| (14) | $T_3$: UNLOCK $A$ | $f_6(f_5(f_1(A_0)))$ | $f_2(f_3(B_0))$ | $f_7(f_4(C_0))$ |

**Fig. 11.4.** A schedule.

Diese Ausführung ist nicht-serialisierbar!

**Beweis:** Finden eines äquivalenten seriellen Schedules.

**Angenommen:**

Wenn T1 vor T2 ist, dann ist der Wert für

$$B = f_3 ( f_2 ( B_0 ) )$$

$$A = f_6 ( f_5 ( f_1 ( A_0 ) ) )$$

$$C = f_7 ( f_4 ( C_0 ) )$$

Wenn T 2 vor T1 ist , dann gilt für A

- T2 T1 T3: $A = f_6 ( f_1 ( f_5 ( A_0 ) ) )$
- T2 T3 T1: $A = f_1 ( f_6 ( f_5 ( A_0 ) ) )$
- T3 T2 T1: $A = f_1 ( f_5 ( f_6 ( A_0 ) ) )$

T2 kann in einem seriellen Schedule werde vor noch nach T1 kommen, also gibt es kein serielles Schedule.

## Ein Protokoll, daß Serialisierbarkeit garantiert

Ein Test aus Serialisierbarkeit ist in der Praxis nicht sehr nützlich, besser ist ein Verfahren, das garantiert, dass alle Schedules serialisierbar sind:

**Zwei-Phasen-Sperrprotokoll (Two phase locking):**

Das Protokoll ist so, dass verlangt wird, dass in einer Transaktion **alle Locks** vor **allen Unlocks** stattfinden.

Eine Transaktion die diesem Protokoll folgt, wird zweiphasig genannt. Die erste Phase ist die Lock-Phase und erst wenn diese beendet ist, kann die zweite Phase, die Unlock-Phase beginnen.

| |
|---|
| Wenn in einem Schedule alle Transaktionen zweiphasig sind, ist es serialisierbar. |

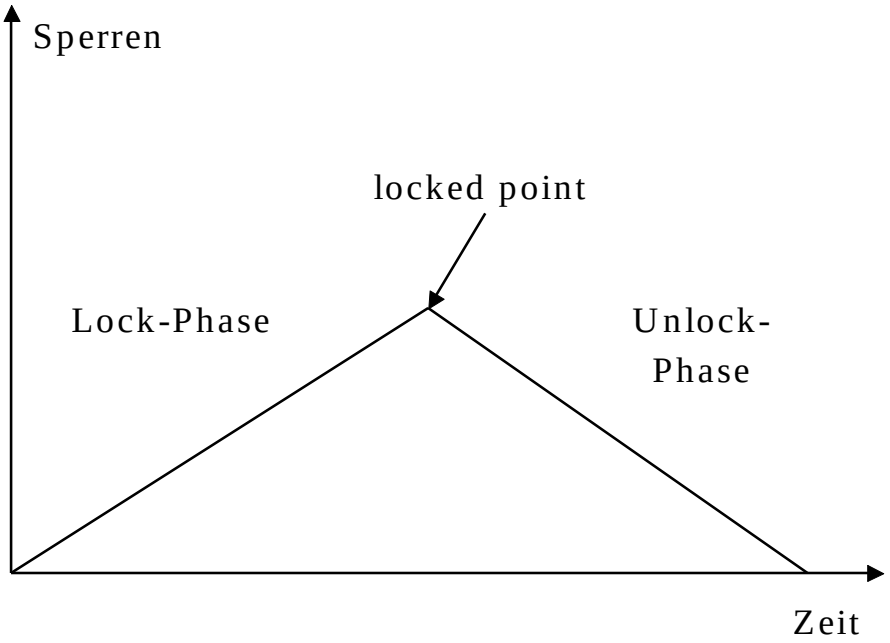
Es gilt also:

Eine Transaktion T=($a_i$) i=1,…,n ist eine 2-Phasen-Transaktion, wenn für ein j < n gilt:

1. i < j → $a_i$ <> UNLOCK
2. i = j → $a_i$ = UNLOCK
3. i > j → $a_i$ <> LOCK

Die Schritte 1, …, j-1 sind die Lock-Phase und die Schritte j, …, n sind die Unlock-Phase der Transaktion. Der Zeitpunkt des Belegens der letzten Sperre wird als **locked point** bezeichnet.

# Diagramm: 2-Phasen-Sperrprotokoll



## Hinreichende Bedingung für Serialisierbarkeit:

Wenn alle Transaktionen einer Transaktionsmenge T wohlgeformt und 2-Phasen-Transaktionen sind, dann ist jede gültige Ausführung von T serialisierbar.

Die Serialisierungsreihenfolge einer solchen Ausführung entspricht der Reihenfolge der locked points der einzelnen Transaktionen aus T.

## Beispiel: 2-Phasen-Sperprotokoll

(A)

```
LOCK a
LOCK b
LOCK c
                ← locked point
UNLOCK c
UNLOCK a
UNLOCK b
```

(B)

```
LOCK a
LOCK b
UNLOCK a
LOCK c      ← nicht 2-Phasen
UNLOCK c
UNLOCK b
```

## Beispiel:

| $T_1$: | | | | $T_2$: | | | |
|---|---|---|---|---|---|---|---|
| | 1 | LOCK | *A* | | 1 | LOCK | *A* |
| | 2 | UPDATE | *A* | | 2 | UPDATE | *A* |
| | 3 | LOCK | *B* | | 3 | LOCK | *B* |
| | 4 | UNLOCK | *A* | | 4 | UPDATE | *B* |
| | 5 | UPDATE | *B* | | 5 | UPDATE | *A* |
| | 6 | UNLOCK | *B* | | 6 | UNLOCK | *A* |
| | | | | | 7 | UPDATE | *B* |
| | | | | | 8 | UNLOCK | *B* |

The following is a legal execution $I_1$ of the transactions $T_1$ and $T_2$.

| $I_1$ | $T_1$: | | | $T_2$: | |
|---|---|---|---|---|---|
| | 1. | LOCK | *A* | | |
| | 2. | UPDATE | *A* | | |
| | 3. | LOCK | *B* | | |
| | 4. | UNLOCK | *A* | | |
| | 5. | | | LOCK | *A* |
| | 6. | | | UPDATE | *A* |
| | 7. | UPDATE | *B* | | |
| | 8. | UNLOCK | *B* | | |
| | 9. | | | LOCK | *B* |
| | 10 | | | UPDATE | *B* |
| | 11 | | | UPDATE | *A* |
| | 12 | | | UNLOCK | *A* |
| | 13 | | | UPDATE | *B* |
| | 14 | | | UNLOCK | *B* |

## Beispiel: (Fortsetzung)

Consider also another example of a legal execution $I_2$ of the given transactions $T_1$ and $T_2$:

| $I_1$ | $T_1$: | | $T_2$: | |
|---|---|---|---|---|
| 1. | | | LOCK | *A* |
| 2. | | | UPDATE | *A* |
| 3. | | | LOCK | *B* |
| 4. | | | UPDATE | *B* |
| 5. | | | UPDATE | *A* |
| 6. | | | UNLOCK | *A* |
| 7. | LOCK | *A* | | |
| 8. | UPDATE | *A* | | |
| 9. | | | UPDATE | *B* |
| 10 | | | UNLOCK | *B* |
| 11 | LOCK | *B* | | |
| 12 | UNLOCK | *A* | | |
| 13 | UPDATE | *B* | | |
| 14 | UNLOCK | *B* | | |

# Deadlock

Dynamische Sperren von Objekten und 2-Phasen Sperrprotokolle führen zu dem Problem des Deadlocks.

Deadlocks müssen entweder von vornherein vermieden oder erkannt und aufgelöst werden.

**Beispiel:** Deadlock

```
T1:          T2:
LOCK A
             LOCK B
(LOCK B)                ← T1 wartet auf B
             (LOCK A)   ← T2 wartet auf A
```

# Ein Modell mit Read- und Write-Locks

Bisheriges Modell:

LOCK A
**CHANGE A**
UNLOCK A

Mögliche Unterscheidung zwischen:

READ und WRITE

# Sperrtypen

Es gibt zwei elementare Typen von Sperren:

- Read-Locks (Shared Locks): nur zum Lesen.

- Write-Locks (Exclusive Locks): zum Updaten.

Kompatibilität zwischen Sperren verschiedener Transaktionen.

| | | Bestehende Sperre | |
|---|---|---|---|
| | | R-LOCK | W-LOCK |
| Sperr-anforderung | R-LOCK | ☺ | ☹ |
| | W-LOCK | ☹ | ☹ |

☺ Verträglich, ☹ Konflikt

Beide Sperren, Read-Lock und Write-Lock werden durch ein UNLOCK wieder aufgehoben.

**Bemerkung:**

Es werden weiterhin keine Write-Only-Locks behandelt. Also Transaktionen bei denen der Wert eines Objektes geschrieben wird, ohne das dieses vorher gelesen wurde.

Ein Write-Lock impliziert also ein Lesen, Verändern und Schreiben eines Objektes.

## Annahmen:

Für jede Transaktion gilt:

- kein Unlock, falls sie keinen R/W-Lock hält
- kein R-Lock, falls sie bereits einen Lock hält.
- kein W-Lock, falls sie bereits einen W-Lock hält.

Ein Write-Lock kann auf ein Objekt gesetzt werden, für das die Transaktion bereits einen Read-Lock hält.

## Äquivalenz zweier Schedules

Zwei Schedules S und S' sind äquivalent, wenn:

- S und S' denselben Wert für jedes Objekt produzieren.
- Jeder Read-Lock der durch eine Transaktion Ti durchgeführt wird, in S und S' an Zeitpunkten ausgeführt wird an denen das gesperrte Objekt den selben Wert hat.

## Das 2-Phasen Sperrprotokoll:

Genau wie bei dem vorherigen Modell mit nur einem Sperrtypen, ist das 2-Phasen Sperrprotokoll, bei dem alle Locks allen Unlocks vorausgehen, ausreichend um Serialisierbarkeit zu garantieren.

Ebenso gilt die Umkehrung, dass jede Transaktion, die nicht dem 2-Phasen Protokoll folgt, mit einer anderen Transaktion in einem nicht-serialisierbaren Schedule ausgeführt werden kann.

# Read-Only, Write-Only Modell

Dieses Modell hat eine andere Semantik als das bisherige Read-Write-Modell.

Wenn eine Transaktion ein Objekt Write-Lockt, liest sie seinen Wert **nicht!**

Zum Lesen des Objektes muß die Transaktion explizit einen Read-Lock auf dem Objekt halten.

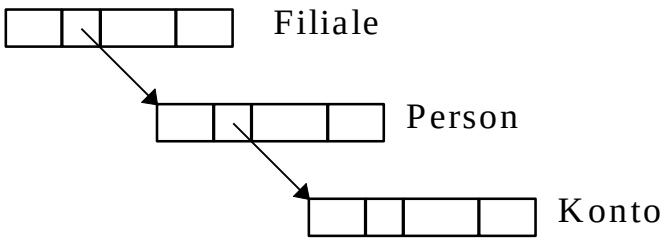
Bei dem bisherigen Modell bezog ein Write-Lock immer die Lese-Berechtigung mit ein. Der Wert des Objektes musste **gelesen und benutzt** werden.

# Nebenläufigkeit bei hierarchisch strukturierten Objekten

Es gibt viele Gelegenheiten, bei denen eine Menge von Objekten natürlicherweise als ein Baum oder Wald angesehen werden können.

**Beispiel:**

1. Die Objekte sind logische Records in einer Datenbank die nach dem hierarchischen Prinzip Modell organisiert ist:




2. Die Objekte sind die Knoten eines B*-Baumes.

3. Die Objekte von verschiedener Größe sind definiert als kleinere Objekte in größeren:

   I. Gesamte Datenbank.

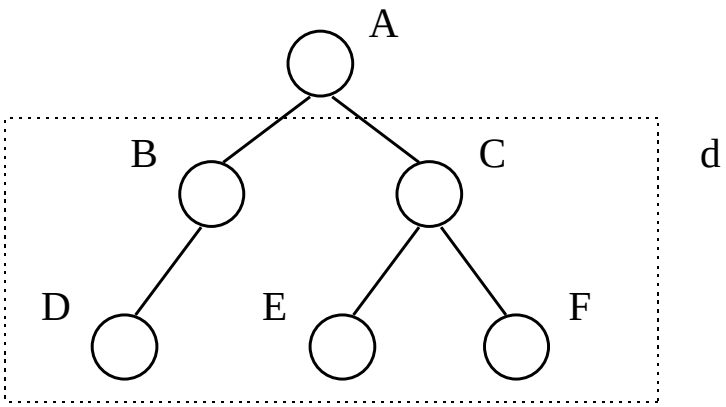
   II. Jede Relation.

   III. Jeder Block einer Datei, die mit einer Relation korreliert.

   IV. Jedes Tupel.

# Zwei Lock-Prinzipien:

1. Ein Lock auf einem Objekt impliziert einen Lock auf allen nachfolgenden Objekten.




LOCK A → LOCK d

2. Locke ein Objekt, ohne irgendeine Auswirkung auf die nachfolgenden Objekte.

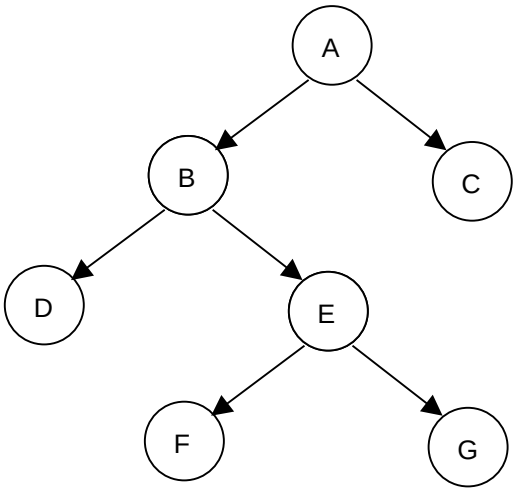
# Baumprotokoll

Eine Transaktion erfüllt das Baumprotokoll, wenn gilt:

1. Ein Objekt A wird nur dann von T gesperrt, wenn T bereits einen Lock auf den Vater von T hält. Dies gilt nicht für das erste Objekt (das muss nicht die Wurzel sein) im Baum, das von T gesperrt wird.

2. Kein Objekt wird von einer Transaktion mehr als einmal gesperrt.

## Beispiel:



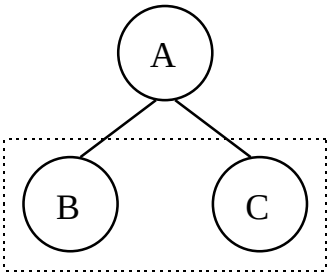

| | $T_1$: | | $T_2$: | | $T_3$: | |
|---|---|---|---|---|---|---|
| 1. | LOCK | *A* | | | | |
| 2. | LOCK | *B* | | | | |
| 3. | LOCK | *D* | | | | |
| 4. | UNLOCK | *B* | | | | |
| 5. | | | LOCK | *B* | | |
| 6. | LOCK | *C* | | | | |
| 7. | | | | | LOCK | *E* |
| 8. | UNLOCK | *D* | | | | |
| 9. | | | | | LOCK | *F* |
| 10 | UNLOCK | *A* | | | | |
| 11 | | | | | LOCK | *G* |
| 12 | UNLOCK | *C* | | | | |
| 13 | | | | | UNLOCK | *E* |
| 14 | | | LOCK | *E* | | |
| 15 | | | | | UNLOCK | *F* |
| 16 | | | UNLOCK | *B* | | |
| 17 | | | | | UNLOCK | *G* |
| 18 | | | UNLOCK | *E* | | |

# Hinreichende Bedingung für Serialisierbarkeit

Wenn alle Transaktionen eines legalen Schedules S dem Baumprotokoll folgen, dann ist S serialisierbar.

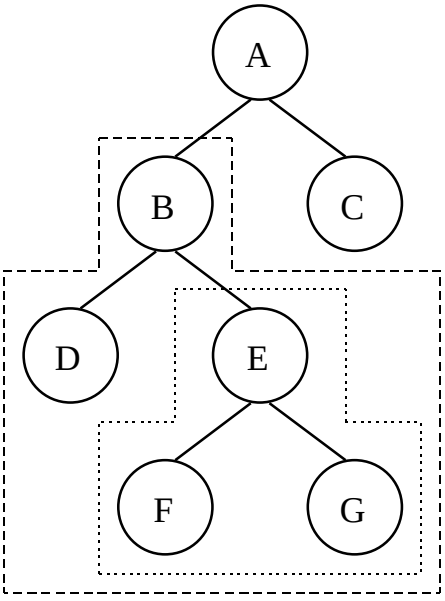
# Sperren auf Subtrees

Einfaches Modell: Sperre alle Nachfolger des Objektes, das gesperrt werden soll.




LOCK A → LOCK A, LOCK B, LOCK C

Wahlloses Sperren von Objekten kann zu illegalen Schedules führen.




T1: LOCK E
T2: LOCK B

zur selben Zeit!

→ Konflikt

Notwendigkeit für ein Protokoll!

# Das hierarchische Sperr-Protokoll

**Informell:** Eine Transaktion kann ein Objekt A nicht sperren, solange sie nicht eine Warnung (Warning) auf alle Vorfahren des Objektes setzt.

Eine Warnung auf A

- verhindert, dass eine andere Transaktion das Objekt A sperrt.
- verhindert keine weitere Warnung auf das Objekt A.
- verhindert nicht das Sperren von Nachfahren des Objektes A, auf denen keine Warnung liegt.

# Modell für Transaktionen

(System R, IBM, San Jose)

Ti:

LOCK (Exclusive, E-Sperre)
Sperrt ein Objekt und alle seine Nachfahren. Zwei Transaktionen können nicht gleichzeitig ein Lock auf ein Objekt halten.

WARN (Intention, I-Sperre)
Keine Transaktion kann ein Objekt sperren, auf dem eine Warnung liegt.

UNLOCK Entfernt ein WARN oder ein LOCK.

Eine Transaktion gehorcht dem Warnungs- oder hierarchischen Sperrprotokoll auf einer Objekt-Hierarchie, wenn sie

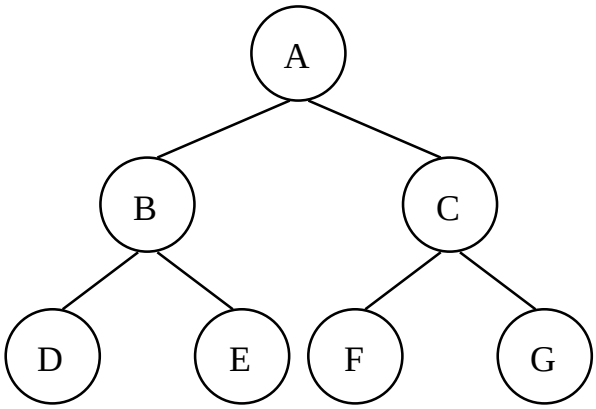
1. damit beginnt, eine Warnung oder Lock auf die Wurzel zu setzen.

2. keine Sperre oder Warnung auf ein Objekt setzt, solange sie keine Warnung auf den Vorfahren (Parent) hält.

3. keine Warnung oder Sperrung von einem Objekt entfernt, solange sie Warnungen oder Sperren auf Nachfahren (Children) des Objektes hält.

4. sie dem 2-Phasen Protokoll folgt.

## Beispiel:



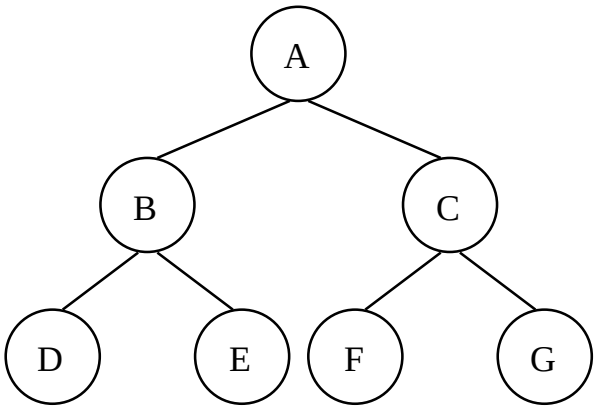

Annahme: Drei Transaktionen die verschiedene Objekte der Hierarchie ändern:

T1: ändert Objekt **D**

T2: ändert Objekt **C**

T3: ändert Objekt **B** und Objekt **F**

Ein legales Schedule für die Ausführung dieser drei Transaktionen wäre z.B. das Folgende:




| | T1 | T2 | T3 |
|---|---|---|---|
| 1 | WARN A | | |
| 2 | | WARN A | |
| 3 | | | WARN A |
| 4 | WARN B | | |
| 5 | | LOCK C | |
| 6 | LOCK D | | |
| 7 | | UNLOCK C | |
| 8 | UNLOCK D | | |
| 9 | | UNLOCK A | |
| 10 | UNLOCK B | | |
| 11 | | | LOCK B |
| 12 | | | WARN C |
| 13 | | | LOCK F |
| 14 | UNLOCK A | | |
| 15 | | | UNLOCK B |
| 16 | | | UNLOCK F |
| 17 | | | UNLOCK C |
| 18 | | | UNLOCK A |

## Beispiel: (Fortsetzung)

- In Schritt 4 setzt T1 eine Warnung auf B, T3 kann also B nicht sperren, bis T1 in Schritt 10 B entsperrt.

- In den Schritten 1,2,3 wird A von allen Transaktionen mit einer Warnung belegt, dies ist legal.

- Die Sperrung von C durch T2 in Schritt 5 sperrt implizit die Objekte C, F und G. Die Annahme ist, das diese Objekte all von T2 geändert werden, bevor die Sperre in Schritt 7 wieder entfernt wird.

## Theorem:

Ausführungen von Transaktionen die dem hierarchischen Sperrprotokoll gehorchen sind serialisierbar.

# OPTIMISTIC CONCURRENCY CONTROL

- Synchronisierung von Transaktionen ohne Sperren.

- Annahme: Es gibt eine Methode, zu erkennen, wann eine Transaktion die serielle Ordnung verletzt.

- Abbruch einer Transaktion und Neustart.

Dieser Ansatz ist sehr effizient, wenn die Wahrscheinlichkeit von Konflikten zwischen Transaktionen gering ist.

→ Granularität der Objekte
(Je kleiner die Objekte sind, auf die zugegriffen wird, desto geringer ist die Wahrscheinlichkeit eines Konfliktes)

Trade-Off zwischen

Zeit, die auf Sperren gewartet wird

und

Zeit, die benötigt wird, eine
Transaktion erneut zu starten.

- Wie kann man erkennen, dass eine Transaktion T die serielle Ordnung verletzt?

- Wie kann man vermeiden, dass in der Datenbank Werte stehen bleiben, die von einer Transaktion T geschrieben wurden, die danach aborted wurde?

- Wie kann ein Live-Lock vermieden werden (Wenn beispielsweise T immer wieder abbricht)?

# Lösungen

Serielle Ordnung

→ Zeitstempel (Timestamps)

„beschädigte“ Werte

→ Commitment

Live-Lock

→ Restart

Betrachten wir das erste Beispiel nicht serialisierbaren Verhaltens:

```
T1:  READ A
T2:  READ A
T1:  A := A + 1
T2:  A := A + 1
T1:  WRITE A
T2:  WRITE A
```

Wir benötigen eine Methode, zu erkennen wann eine Transaktion die serielle Ordnung verletzt, um dann die Transaktion neu zu starten.

| A | 5 | 5 | 5 | 5 | 6 | 6 |
|---|---|---|---|---|---|---|
| $T_1$ | READ A | | A:=A+1 | | | WRITE A |
| $T_2$ | | READ A | | A:=A+1 | WRITE A | |
| $A(T_1)$ | 5 | 5 | 6 | 6 | 6 | 6 |
| $A(T_2)$ | | 5 | 5 | 6 | 6 | |

# Zeitstempelverfahren (Time stamping)

Um die serielle Ordnung aufrecht zu erhalten und Verletzungen der seriellen Ordnung erkennen zu können, generiert das System Zeitstempel.

- Eine Zahl, die bei jedem "Tick" der Systemuhr generiert wird (z.B. jede Mikrosekunde).

- Ein 32-bit Wort reicht, um die Anzahl von "Ticks" eines ganzen Jahrhunderts aufzunehmen.

- Eine solche Zahl wird jeder Transaktion so zugeordnet, daß keine zwei Transaktionen den selben Zeitstempel haben. (Beginn der Transaktion)

- Mit jedem Objekt der Datenbank werden zwei Zeitstempel gespeichert:
  - $t_r$ = read-time (Lese-Zeitstempel)
    Der Zeitstempel der Transaktion mit dem höchsten (d.h. spätesten) Zeitstempel von allen Transaktionen, die das Objekt **gelesen** haben.
  - $t_w$ = write-time (Schreib-Zeitstempel)
    Der Zeitstempel der Transaktion mit dem höchsten (d.h. spätesten) Zeitstempel von allen Transaktionen, die das Objekt **geschrieben** haben.
- Wie kann man Transaktionen nun dazu bringen, sich so zu verhalten, als ob sie seriell abgelaufen wären?

## Korrektheit:

Transaktionen sollten sich so verhalten, als ob die Ordnung ihrer Zeitstempel die serielle Ordnung gewesen wäre.

Durch folgende Bedingungen wird die Konsistenz der Datenbank sichergestellt:

1. Eine Transaktion T darf ein Objekt nicht lesen, wenn der Wert des Objektes von einer Transaktion geschrieben wurde, die (in der seriellen Ordnung) nach ihr stattfindet:

> Eine Transaktion T mit Zeitstempel $t_1$ darf ein Objekt mit Schreib-Zeitstempel $t_w$ nicht lesen, wenn $t_w > t_1$ gilt.
>
> T muss dann abgebrochen (**roll-back**) und neu gestartet (**restart**) werden.

2. Eine Transaktion T darf ein Objekt nicht schreiben, wenn der alte Wert des Objektes bereits von einer Transaktion gelesen wurde, die (in der seriellen Ordnung) nach ihr stattfindet:

> Eine Transaktion T mit Zeitstempel $t_1$ darf ein Objekt mit Lese-Zeitstempel $t_r$ nicht schreiben, wenn $t_r > t_1$ gilt.
>
> T muss dann abgebrochen und neu gestartet werden

- Zwei Transaktionen können das gleiche Objekt zu verschiedenen Zeiten lesen. T mit Zeitstempel $t_1$ kann ein Objekt mit Zeitstempel $t_r$ lesen, selbst wenn $t_r > t_1$ ist.

- Eine Transaktion T mit Zeitstempel $t_1$ muss nicht abbrechen, wenn sie versucht ein Objekt mit Zeitstempel $t_w$ zu schreiben, selbst wenn $t_w > t_1$ ist. Der Wert wird in diesem Fall einfach nicht geschrieben.

  Begründung:

  In der seriellen Reihenfolge basierend auf den Zeitstempeln, hätte erst die Transaktion mit Zeitstempel $t_1$ und anschließend die Transaktion mit Zeitstempel $t_w$ das Objekt geschrieben.

  Zu beachten ist, dass zwischen $t_1$ und $t_w$ das Objekt nicht gelesen werden darf, ansonsten wäre $t_r$ des Objektes größer als $t_1$ und die Transaktion T müsste nach Regel 2 abgebrochen werden.

## Beispiel:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| A<br>tr=0<br>tw=0 | 5<br>tr=150<br>tw=0 | 5<br>tr=150<br>tw=0 | 5<br>tr=150<br>tw=0 | 12<br>tr=150<br>tw=160 | 12<br>tr=150<br>tw=160 |
| $T_1$<br>$t_1$=150 | READ A | A:=A+1 | | | WRITE A |
| $T_2$<br>$t_2$=160 | | | A:=12 | WRITE A | |

- Eine abgebrochene Transaktion erhält beim Restart einen neuen Zeitstempel.

- Die Serialisierungsreihenfolge der Transaktionen entspricht der Reihenfolge der Zeitstempel, also der Startzeitpunkte.

# Algorithmus

Die Transaktion T mit Zeitstempel t will eine Operation X auf dem Objekt A mit den Zeitstempeln $t_r$ und $t_w$ ausführen.

1. if X = READ and t >= $t_w$ then
   READ A
   if t > $t_r$ then $t_r$ := t

2. if X = WRITE and t >= $t_r$ and t >= $t_w$ then
   WRITE A
   $t_w$ := t

3. if X = WRITE and $t_r$ <= t < $t_w$ then
   { do nothing }

4. if ( X = READ and t < $t_w$ ) or
   (X = WRITE and t < $t_r$ ) then
   abort(T)

## Beispiel:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| A<br>tr=0<br>tw=0 | 5<br>tr=150<br>tw=0 | 5<br>tr=160<br>tw=0 | 5<br>tr=160<br>tw=0 | 5<br>tr=160<br>tw=0 | 6<br>tr=160<br>tw=160 | 6<br>tr=160<br>tw=160 |
| $T_1$<br>$t_1$=150 | READ A | | A:=A+1 | | | WRITE A |
| $T_2$<br>$t_2$=160 | | READ A | | A:=A+1 | WRITE A | |
| $A(T_1)$ | 5 | 5 | 6 | 6 | 6 | 6 |
| $A(T_2)$ | | 5 | 5 | 6 | 6 | |

Schritt 1: $t_1 > t_r \rightarrow T_1$ liest A; $t_r := 150$

Schritt 2: $t_2 > t_r \rightarrow T_2$ liest A; $t_r := 160$

Schritt 3: keine Datenbankoperation

Schritt 4: keine Datenbankoperation

Schritt 5: $t_2 \geq t_w$ und $t_2 \geq t_r \rightarrow T_2$ schreibt A; $t_w := 160$

Schritt 6: $t_1 < t_r \rightarrow T_1$ wird abgebrochen !

## Beispiel:

| | T1 | T2 | T3 | A | B | C |
|---|---|---|---|---|---|---|
| | 200 | 150 | 175 | RT = 0 | RT = 0 | RT = 0 |
| | | | | WT = 0 | WT = 0 | WT = 0 |
| (1) | READ B | | | | RT = 200 | |
| (2) | | READ A | | RT = 150 | | |
| (3) | | | READ C | | | RT = 175 |
| (4) | WRITE B | | | | WT = | |
| (5) | WRITE A | | | WT = | | |
| (6) | | WRITE C | | | | |
| (7) | | | WRITE A | | | |

Fig. 11.20. Optimistically executing transactions.

Schritt 6: Das Objekt C soll von $T_2$ geschrieben werden; der Zeitstempel von $T_2$ ist 150, $t_r$ von C ist aber 175; $T_2$ muss also abgebrochen werden.

Schritt 7: $T_3$ versucht, Objekt A zu schreiben; der Zeitstempel von $T_3$ ist mit 175 kleiner als der Schreib-Zeitstempel von A; $T_3$ muß nicht abgebrochen werden, aber der Wert den $T_3$ schreiben wollte, wird nicht in der Datenbank eingetragen.

## Commitment

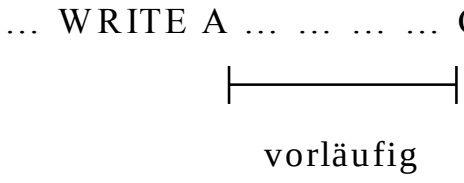
- Es muss in Betracht gezogen werden, dass eine Transaktion, die abgebrochen wird, einen Wert in die Datenbank geschrieben haben kann.

Lösung:

- WRITE verändert keine Werte in der Datenbank.
- Der Wert wird in ein "Journal" geschrieben (das Logfile).
- $t_w$ wird nur vorläufig verändert.
- der alte **und** der neue Wert von $t_w$ werden in das Journal aufgenommen.
- Nachdem T alle WRITES beendet hat, führt sie ein COMMIT aus.
  → alle geschriebenen Werte werden jetzt permanent in der Datenbank gespeichert.

- In der Zeit zwischen dem WRITE und dem COMMIT darf keine Transaktion den neuen Wert des Objektes lesen, da er nur vorläufig ist.

- Eine Transaktion die diesen Wert gelesen hat, müsste sonst ebenfalls abgebrochen werden, wenn T abgebrochen wird.




- Es ist deshalb gut, wenn der Abstand zwischen dem WRITE und dem COMMIT kurz ist.

- Eine Transaktion mit einem höheren Zeitstempel darf das Objekt schreiben und überschreibt dadurch den vorläufigen Wert des Objektes (mit einen ebenfalls vorläufigen Wert).

- Wenn der alte Wert und der alte Schreib-Zeitstempel $t_w$ in der Datenbank verfügbar gehalten werden, können READ-Anfragen von Transaktionen mit Zeitstempeln zwischen dem alten und dem neuen Schreib-Zeitstempel mit dem alten Wert des Objektes bedient werden.

# Vergleich Optimistic vs. Locking

Die Anzahl der Datenbankoperationen ist bei der Optimistic Concurrency Control kleiner als beim Locking.

Im Schlimmsten Fall (dem Update) sind für Optimistic Concurrency Control 2 Operationen nötig, für das Locking aber 4:

| **Optimistic** | **Locking** |
|---|---|
| WRITE | LOCK |
| COMMIT | WRITE |
| | UNLOCK |
| | COMMIT |

Wenn Konflikte zwischen Transaktionen nicht sehr häufig sind, ist der Ansatz der Optimistic Concurrency Control dem Locking vorzuziehen.

# Restart von Transaktionen

**Problem:** Live-Lock

Eine Transaktion wird immer wieder abgebrochen.

## Beispiel:

| | T1 | T2 | T1 | T2 | A | B |
|---|---|---|---|---|---|---|
| | 100 | 110 | 120 | 130 | RT = 0 | RT = 0 |
| | | | | | WT = 0 | WT = 0 |
| (1) | WRITE B | | | | | WT = 100 |
| (2) | | WRITE A | | | WT = | |
| (3) | READ A | | | | | |
| (4) | | | WRITE B | | | WT = 120 |
| (5) | | READ B | | | | |
| (6) | | | | WRITE A | WT = | |
| (7) | | | READ A | | | |

Fig. 11.21. Indefinite repetition of two confliction transactions.

**<u>Beispiel</u>: (Fortsetzung)**

Schritt 3: $T_1$ will A lesen, A hat aber einen Schreib-Zeitstempel der größer ist als der Zeitstempel von $T_1$. $T_1$ wird daher abgebrochen.

$T_1$ wird sofort wieder mit einem Zeitstempel von 120 gestartet.

Schritt 4: $T_1$ schreibt B → $B(t_w)$ := 120

Schritt 5: $T_2$ versucht B zu lesen → abort.

$T_2$ wird sofort wieder mit einem Zeitstempel von 130 gestartet.

Dies kann sich immer weiter wiederholen.

Es gibt für dieses Problem keine einfache Lösung.

→ Benutze einen Zufallszahlengenerator, um eine Zeit $\Delta t$ zu ermitteln, die eine abgebrochene Transaktion warten muss, bis sie wieder gestartet werden kann.

**Notiz:** Auch bei dieser Lösung kann es theoretisch passieren, dass ein Live-Lock eintritt, allerdings mit einer geringeren Wahrscheinlichkeit.